ЛЕО БРОУДИ

СПОСОБ МЫШЛЕНИЯ -

Ф О Р Т

ЯЗЫК И ФИЛОСОФИЯ

ДЛЯ РЕШЕНИЯ ЗАДАЧ

Leo Brodie

Thinking FORTH
A Language and Philosophy for Solving Problems

Englewood Cliffs, N.J., Prentice-Hall, Inc., 1984

Перевод с английского С.Н.Дмитренко
(Москва, 1993 г.)

Лео Броуди - писатель, программист и консультант, авторитет мирового масштаба в области языка программирования Форт. Он был техническим программистом в фирме FORTH, Inc. а с 1981 года стал независимым консультантом для фирм IBM, NCR и Lockheed. Он является также автором книги Starting FORTH, Prentice-Hall, 1981 (перевод: "Начальный курс программирования на языке Форт", М: Финансы и статистика, 1990).

Посвящается Стефани, Брэндому и Раяну.

"Невозможно отделить язык от науки или науку от языка, поскольку любая естественная наука всегда использует три вещи: последовательность феноменов, на которые она опирается, краткие описания - концепции этих феноменов, которыми они представляются в мышлении, и слова, в которых выражаются концепции. Для движения к концепции необходимо слово; для описания явления необходима концепция. Все три отражают одну и ту же реальность."

Антони Лавуазье, 1789.

СОДЕРЖАНИЕ

СПИСОК ПРИМЕРОВ ПРОГРАММ 5
ОТ ПЕРЕВОДЧИКА 6
ПРЕДИСЛОВИЕ 8
БЛАГОДАРНОСТИ 9
ГЛАВА 1 ФИЛОСОФИЯ ФОРТА 10
СКАЗАНИЕ ОБ ИСТОРИИ ЭЛЕГАНТНОСТИ ПРОГРАММ 10
ПОВЕРХНОСТНОСТЬ СТРУКТУРЫ 24
ВЗГЛЯД НАЗАД, ВПЕРЕД И НА ФОРТ (*) 25
ПРОГРАММИРОВАНИЕ НА УРОВНЕ КОМПОНЕНТОВ 27
ОТ КОГО ПРЯТАТЬ? 30
УПРЯТЫВАНИЕ КОНСТРУКЦИИ СТРУКТУР ДАННЫХ 31
НО ВЫСОКОУРОВНЕВЫЙ ЛИ ЭТО ЯЗЫК? 33
ЯЗЫК ПРОЕКТИРОВАНИЯ 36
ПРОИЗВОДИТЕЛЬНЫЙ ЯЗЫК 36
ИТОГИ 39
ЛИТЕРАТУРА 40
ГЛАВА 2 АНАЛИЗ 41
ДЕВЯТЬ ФАЗ ЦИКЛА ПРОГРАММИРОВАНИЯ 41
ИТЕРАТИВНЫЙ ПОДХОД 42
ОБЬЕМ ПЛАНИРОВАНИЯ 43
ОГРАНИЧЕНИЯ ПЛАНИРОВАНИЯ 45
ФАЗА АНАЛИЗА 48
ОПРЕДЕЛЕНИЕ ИНТЕРФЕЙСОВ 51
ОПРЕДЕЛЕНИЕ ПРАВИЛ 55
ОПРЕДЕЛЕНИЕ СТРУКТУР ДАННЫХ 61
ДОСТИЖЕНИЕ ПРОСТОТЫ 62
СОБЛЮДЕНИЕ БЮДЖЕТА И ГРАФИКА 67
СМОТРИНЫ ДЛЯ КОНЦЕПТУАЛЬНОЙ МОДЕЛИ 69
ЛИТЕРАТУРА 69
ГЛАВА 3 ПРЕДВАРИТЕЛЬНЫЙ ПРОЕКТ / ДЕКОМПОЗИЦИЯ 70
ДЕКОМПОЗИЦИЯ ПО КОМПОНЕНТАМ 70
ПРИМЕР: КРОШЕЧНЫЙ РЕДАКТОР 72
ПОДДЕРЖКА ЗАДАЧИ, ОСНОВАННОЙ НА КОМПОНЕНТАХ 76
ПРОЕКТИРОВАНИЕ И ПОДДЕРЖКА ЗАДАЧИ 77
ПРИ ТРАДИЦИОННОМ ПОДХОДЕ 77
ИНТЕРФЕЙСНЫЙ КОМПОНЕНТ 82
РАЗБИЕНИЕ ПО ПОСЛЕДОВАТЕЛЬНЫМ УРОВНЯМ СЛОЖНОСТИ 85
ОГРАНИЧЕННОСТЬ МЫШЛЕНИЯ ПО УРОВНЯМ 87
РЕЗЮМЕ 90
ДЛЯ ДАЛЬНЕЙШЕГО РАЗМЫШЛЕНИЯ 91
ГЛАВА 4 ДЕТАЛИЗИРОВАННАЯ РАЗРАБОТКА РЕШЕНИЕ ЗАДАЧИ 94
ТЕХНИКА РЕШЕНИЯ ЗАДАЧ 94
ИНТЕРВЬЮ С ИЗОБРЕТАТЕЛЕМ-ПРОГРАММИСТОМ 100
ДЕТАЛИЗИРОВАННАЯ РАЗРАБОТКА 102
СИНТАКСИС ФОРТА 103
АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ 111
РАСЧЕТЫ ИЛИ СТРУКТУРЫ ДАННЫХ ИЛИ ЛОГИКА 112
РЕШЕНИЕ ЗАДАЧИ: ВЫЧИСЛЕНИЕ РИМСКИХ ЦИФР 113
ИТОГИ 126
ЛИТЕРАТУРА 126

ДЛЯ ДАЛЬНЕЙШИХ РАЗМЫШЛЕНИЙ 126
ГЛАВА 5 РАЗРАБОТКА : ЭЛЕМЕНТЫ ФОРТ - СТИЛЯ 127
ОРГАНИЗАЦИЯ ЛИСТИНГОВ 127
ДЕКОМПОЗИЦИЯ. 129
ОФОРМЛЕНИЕ БЛОКА 137
СОГЛАШЕНИЯ ПО КОММЕНТАРИЯМ 141
ВЕРТИКАЛЬНЫЙ ФОРМАТ ЗАПИСИ ПРОТИВ ГОРИЗОНТАЛЬНОГО. 151
ВЫБОР ИМЕН: ИСКУССТВО 154
СТАНДАРТЫ ПРИ ВЫБОРЕ ИМЕН: НАУКА 158
ЕЩЕ СОВЕТЫ ПО ЧИТАБЕЛЬНОСТИ 160
ИТОГИ 161
ЛИТЕРАТУРА 161
ГЛАВА 6. Ф Р А Г М Е Н Т А Ц И Я 162
ТЕХНИКА ФАКТОРИЗАЦИИ 162
КРИТЕРИИ ДЛЯ ФРАГМЕНТАЦИИ 168
ФАКТОРИЗАЦИЯ ПРИ КОМПИЛЯЦИИ 178
ИТЕРАТИВНЫЙ ПОДХОД ПРИ РЕАЛИЗАЦИИ 182
ИТОГИ 185
ЛИТЕРАТУРА 185
ГЛАВА 7 РАБОТА С ДАННЫМИ : СТЕКИ И СОСТОЯНИЯ 186
ШИКАРНЫЙ СТЕК 186
ШИКАРНЫЙ СТЕК ВОЗВРАТОВ 194
ПРОБЛЕМА ПЕРЕМЕННЫХ 194
ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ / ИНИЦИАЛИЗАЦИЯ 197
СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ СОСТОЯНИЯ 198
ВНУТРЕННИЕ СТЕКИ ПРОГРАММ 200
СОВМЕСТНОЕ ИСПОЛЬЗОВАНИЕ КОМПОНЕНТОВ 201
ТАБЛИЦА СОСТОЯНИЯ 202
ВЕКТОРИЗОВАННОЕ ИСПОЛНЕНИЕ 206
ИСПОЛЬЗОВАНИЕ DOER/MAKE 209
ИТОГИ 213
ЛИТЕРАТУРА 213
ГЛАВА 8 МИНИМИЗАЦИЯ СТРУКТУР УПРАВЛЕНИЯ 214
ЧТО ЖЕ ТАКОГО ПЛОХОГО В СТРУКТУРАХ УПРАВЛЕНИЯ? 214
КАК УСТРАНЯТЬ СТРУКТУРЫ УПРАВЛЕНИЯ 218
ИТОГИ 244
ЛИТЕРАТУРА 245
ДЛЯ ДАЛЬНЕЙШИХ РАЗМЫШЛЕНИЙ 245
ПРИЛОЖЕНИЕ А ОБЗОР ФОРТА (ДЛЯ НОВИЧКОВ) 248
ПРИЛОЖЕНИЕ Б ОПРЕДЕЛЕНИЕ DOER / MAKE 251
ПРИЛОЖЕНИЕ В ДРУГИЕ УТИЛИТЫ , ОПИСАННЫЕ В ЭТОЙ КНИГЕ 256
ПРИЛОЖЕНИЕ Г ОТВЕТЫ НА ЗАДАЧИ " ДЛЯ ДАЛЬНЕЙШЕГО РАЗМЫШЛЕНИЯ " 258
ПРИЛОЖЕНИЕ Д СВОД СТИЛИСТИЧЕСКИХ СОГЛАШЕНИЙ 260

СПИСОК ПРИМЕРОВ ПРОГРАММ

ПРОГРАММА	N СТРАНИЦЫ
Яблоки 24 - 26	
Телефонные тарифы	
Крошечный редактор	
Цвета	
Римские числа	
Рисование квадратиков	
Банкомат	

ОТ ПЕРЕВОДЧИКА

Уже несколько лет я использую для программирования язык Форт. С первой же встречи с ним я был очарован и покорен его простотой, элегантностью и логичностью. И Форт пока ни разу не давал мне повода для разочарований. К сожалению, в нашей стране Форт знают и используют лишь считанные энтузиасты, чему в большой мере способствует отсутствие сколько-нибудь доступной литературы и программного обеспечения. Можно сказать много грустных слов о тенденциозности современной околотехнической литературы, о переориентации отечественных программистов и разработчиков с исследовательских и новаторских на чисто коммерческие работы, о складывающемся монополизме отнюдь не лучших (зато более хватких) производителей компьютеров и программного обеспечения и т.д. И все же, несмотря на эти признаки нарастающего вырождения в нашем, да и мировом компьютерном деле, я надеюсь, что подъем наступит, и мы будем его свидетелями и реализаторами.

Впрочем, когда я говорю о недоступности литературы по Форту, я несколько преувеличиваю. Несколько книг все же изданы, и среди них - несколько очень хороших. Я выделил бы такие:

- Баранов С.Н., Ноздрунов Н.Р. Язык Форт и его реализации. - Л.: Машиностроение. Ленингр. отд-ние, 1988.
- Таунсенд К., Фохт Д. Проектирование и программная реализация экспертных систем на персональных ЭВМ. - М.: Финансы и статистика, 1990.
- Броуди Л. Начальный курс программирования на языке Форт. - М.: Финансы и статистика, 1990.

Последняя из этих книг, на мой взгляд - прекрасный учебник по Форту (оригинальное название ее - "Starting FORTH", т.е. "Начала Форта"). Именно знакомство с ней побудило меня искать и вторую книгу того же автора, Лео Броуди, служащую логическим продолжением первой: "Thinking FORTH" (я перевел это название как "Способ мышления - Форт", хотя игру слов на русский язык точно перевести, кажется, невозможно). Мне не удалось найти ее перевода, и поэтому я обратился к копии оригинала. Впоследствии выяснилось, что в С.-Петербурге перевод этой книги имеется, издание же его (которое должно было последовать вслед за "Начальным курсом...") пока не состоялось из-за известных финансовых затруднений.

Мне кажется, книг, подобных "Thinking FORTH", в нашей стране не видели. По крайней мере, столь подробного и приближенного к подлинной практике описания процесса создания программного обеспечения - и при этом без традиционного наукообразия - мне раньше читать не доводилось. Даже если предположить, что лишь один я столь мало эрудирован, это все равно нисколько не может умалить интересность книги для любого увлеченного программиста - и необязательно только фортиста. Я просто не мог держать эту книгу только для себя - и рискнул выполнить собственный ее перевод для тех, кому английский оригинал недоступен.

Я не литератор и не профессиональный переводчик, поэтому приношу извинения за порою встречающиеся англицизмы и коряво построенные русские фразы. Будем считать этот вариант черновым и ознакомительным, я надеюсь только, что правильно передал
смысл текста.

Перевод готовился мною намеренно в самом простом формате - в виде компьютерных ASCII-файлов. Это позволяет читать и распечатывать текст практически на любых компьютерах и принтерах (расчитывать на хорошо обеспеченных читателей пока не приходится). Платой за это служит исключение из текста многочисленных

остроумных авторских иллюстраций и рисунков, которые слишком затруднительно было бы отображать алфавитно-цифровыми символами. Все же я постарался сохранить все существенные для понимания сути картинки.

Я старался держать перевод как можно ближе к тексту оригинала, однако все же позволил себе сделать несколько, на мой взгляд, существенных, правок и ремарок. В наибольшей степени это касается употребления термина "блок" вместо использовавшегося Броуди термина "экран" - это сделано в соответствии с рекомендациями из "Начального курса...", который вышел значительно позже имеющегося у меня оригинала "Thinking FORTH" (от 1984 года). Кроме того, с целью улучшения восприятия в книге переведены на русский язык имена слов, использующихся для иллюстраций и примеров. Кстати, следует иметь в виду, что лишь некоторые реальные Форт-системы (будучи рожденными вне России) допускают использование русских букв в именах определений (заметим, что для классических языков программирования о такой возможности даже речи нет). Пример Форт-системы, в которой можно использовать для имен любые символы - FORTH-83 V4.0 Undirect (&-C, piton DS @1992) для компьютеров класса PDP-11 (RT-11: ДВК, СМ, БК, УКНЦ и т.д.).

Надеюсь, что эта книга поможет Вам лучше понять суть работы программиста-практика, расширить свой кругозор и очистить мышление. Броуди высказывает множество интересных и порою спорных мыслей, поднимает темы, которые весьма актуальны и сегодня - особенно в связи с "непрограммистскими" тенденциями в программировании, переусложненностью современных систем и борьбой за сохранение контроля над такими системами. Интересно проследить точки соприкосновения и различия между модным сегодня объектно-ориентированным программированием (по-моему, способом улучшения понимания системы человеком путем внесения огромной избыточности в исполняющую систему или компилятор) и Форт-методом (достижения того же путем исключения избыточности и достижения простоты). Форт дает совершенно иное решение "вечных" проблем программирования, позволяет избавиться от страха перед большими системами и задачами. Мне кажется, распространению Форта мешают лишь два фактора:

1) воспитываемая сызмальства "классическая" методология мышления у программистов (мешающая в равной степени и внедрению объектно-ориентированной технологии) и
2) направленность Форта на создание "конечных" продуктов и систем, особенно - специализированных, и ограниченность (минимальность) имеющихся стандартов. Если вторая проблема вполне преодолима, то первая будет еще многие годы тянуть назад всю компьютерную науку (как она и тянет ее уже не первое десятилетие).

Постарайтесь освободиться от консервативного скепсиса; не обязательно целиком принимать Форт или какие-то размышления Броуди, достаточно хотя бы усомниться в незыблемости устоявшихся и непреложности модных течений в искусстве программирования. Наградой послужит уже то, что Ваше мышление будет подготовлено к восприятию перемен, которые, несомненно, принесет нам XXI век.

С.Н.Дмитренко, 8 апреля 1993 г.

ПРЕДИСЛОВИЕ

Программирование компьютеров может свести с ума. Другие профессии дают Вам прекрасные возможности наблюдать осязаемые результаты Ваших усилий. Часовщик может смотреть на свои зубчики и колесики, швея - на швы, ровно ложащиеся после каждого взмаха иглы. Но программист проектирует, строит и ремонтирует нечто воображаемое, призрачные механизмы, ускользающие от восприятия органами чувств. Наша работа происходит не в ОЗУ, не в программе-редакторе, а внутри нашей головы.

Построение моделей в воображении привлекает и доставляет удовольствие программисту. Как же лучше к этому подготовиться? Вооружиться самыми хорошими отладчиками, декомпиляторами и дизассемблерами? Они помогают, однако самые существенные из технологий и инструментов - умственные. Нам нужна последовательная и практическая методология для `мышления` на тему задач программирования. Это и составляет суть того, что я попытался выразить в моей книге. "Способ мышления ..." предлагается всем, кто заинтересован в написании программ для решения конкретных задач. Книга рассматривает вопросы проектирования и применения: принятие решений о том, что Вам нужно сделать, разработка компонентов системы и, наконец, построение системы.

В книге подчеркивается важность написания программ не просто работоспособных, но и надежных, логичных и выражающих наилучшее решение проблемы самыми простыми методами.

Несмотря на то, что описываемые здесь принципы могут быть применены к любому языку, я представил их в контексте языка Форт. Форт - это язык, операционная система, набор инструментов и философия. Это - идеальное средство для мышления, поскольку оно соответствует тому способу, по которому работают наши головы. Думать на Форте значит думать просто, думать элегантно, думать гибко. Такое мышление `не` имеет запретительного характера, `не` сложно, `не` чрезмерно теоретизировано. Вам даже не нужно знать Форт для получения пользы от этой книги. Книга "Способ мышления - Форт" сочетает Форт-метод со многими принципами, выработанными современной компьютерной наукой. Союз между простотой Форта и традиционной дисциплиной анализа и стилистки даст Вам новый и лучший способ подхода к задачам программирования и окажет помощь во всех областях применения компьютеров.

Если Вы хотите узнать больше о Форте, другая моя книга - "Начальный курс программирования на языке Форт" – содержит сведения об этом языке. Кроме того, такие сведения приводятся в приложении А данной книги.

Несколько слов о плане этой книги. Первая глава посвящена основным соображениям, далее я провел книгу по основному циклу создания программного обеспечения: от начальных требований до внедрения. Приложения в конце включают обзор Форта для тех, кто с ним не знаком, тексты для нескольких описанных в книге программ, ответы на вопросы и свод соглашений по стилистике.

Многие мысли в этой книге не являются научными. Они основаны на субъективном опыте и наблюдениях за самим собой. По этой причине я привел в книге интервью с большим количеством профессионалов, работающих на Форте, и не все из них полностью согласны друг с другом или со мной. Все эти мнения могут изменяться изготовителем без специального уведомления. В книге вносятся также предложения, называемые "советами". Подразумевается, что им следует внимать лишь тогда, когда они соответствуют Вашей ситуации. В Форт-мышлении нет нерушимых правил. Для обеспечения возможно большего соответствия возможным Форт-системам все примеры программ в книге соответствуют стандарту Форт-83.

Личность, в сильной степени повлиявшая на эту книгу – это человек, придумавший Форт - Чарльз Мур. В дополнение к нескольким дням, проведенным за

интервьюированием его для книги, я имел возможность понаблюдать его за работой. Он - хозяин своего дела, двигающийся в нем быстро и искусно так, как будто он физически реализует концептуальные модели внутри машины - строя, оттачивая, обыгрывая. Он обходится минимумом инструментов (результат продолжающейся борьбы против внутренней сложности) и немногими ограничениями, дополняющими те, которые накладываются его собственной технологией. Я надеюсь, что эта книга уловила что-то из его мудрости. Пользуйтесь!

БЛАГОДАРНОСТИ

Особая признательность всем добрым людям, которые отдали свое время и свои идеи этой книге, среди них: д-р Марк Бернстейн, Дональд Барджисс, Кери Кемпбелл, д-р Раймонд Десси, Том Даулинг, Майкл Хэм, Ким Харрис, Дейв Джонсон, д-р Питер Кожж, Майкл ЛаМанна, Чарльз Х. Мур, д-р Майкл Старлинг и Джон Телеска.

Спасибо также Джерри Бутеллю, Джеймсу Элфу, Джиму Флорною, фирме Moore Products Co. и Карен Нельсон.

Печально, что Майкл ЛаМанна покинул этот мир в то время, когда книга еще писалась. Его глубоко не хватает нам, тем, кто любил его.

ГЛАВА 1

ФИЛОСОФИЯ ФОРТА

Форт является языком и операционной системой. Но это не все: он также и воплощение философии. Обычно философию не рассматривают как нечто, отдельное от Форта. Она не предшествовала Форту и не описывалась где-либо вне рассуждений о Форте, и даже не имеет другого имени, кроме как "Форт".

Что она такое? Как ее можно применять для решения задач программирования? Перед тем, как ответить на эти вопросы, давайте сделаем 100 шагов назад и изучим некоторые из важнейших философий, развитых учеными-компьютерщиками на протяжении многих лет. Проследив траекторию этих достижений, мы сравним - и отделим - Форт от этих принципов в программировании.

СКАЗАНИЕ ОБ ИСТОРИИ ЭЛЕГАНТНОСТИ ПРОГРАММ

В доисторические программные времена, когда компьютеры были еще динозаврами, простой факт того, что некий гений создал программу, которая правильно работает, вызывал великое удивление. По мере роста цивилизованности ЭВМ это удивление слабело. Руководители желали большего от программистов и их программ.

В то время, как стоимость аппаратуры устойчиво падала, стоимость программного обеспечения взмывала ввысь. Для программы было уже недостаточно хорошо просто правильно выполняться. Она должна была быть разработана быстро и обладать легкостью в управлении. Новое качество наряду с корректностью стало важнейшим. Это недостающее качество было названо "элегантностью".

В даном разделе мы проследим историю инструментария и технологий, предназначенных для написания более элегантных программ.

ЗАПОМИНАЕМОСТЬ.

Первые программы для ЭВМ выглядели как-то вроде:

```
00110101
11010011
11011001
```

Программисты вводили их, устанавливая ряды переключателей в положение "вкл." для единиц и "выкл." для нулей. Эти значения были "машинными инструкциями" для ЭВМ, и каждая заставляла ее производить некие приземленные операции типа "переместить содержимое регистра А в регистр Б" или "добавить содержимое регистра В к содержимому регистра А".

Это оказалось несколько скучноватым. Скука - мачеха изобретения, поэтому некоторые умные программисты осознали, что машину и саму можно затставить помочь им. Так они написали программу, которая переводила легкозапоминаемые аббревиатуры в труднозапоминаемые последовательности битов. Новый язык выглядел примерно так:

```
MOV B,A
ADD C,A
JMC REC1
```

Переводчик (транслятор) программ был назван `ассемблером`, а новый язык - `языком ассемблера`. Каждая инструкция "собирала" ("ассемблировала") соответствующую последовательность битов для себя при сохранении точного соотношения между ассемблерной инструкцией и машинной командой. Но ведь имена программистам запоминать легче. По этой причине новые инструкции были названы `мнемониками`.

МОЩНОСТЬ.

Программирование на языке ассемблера характеризуется соответствием "один-в-один" между каждой командой, которую набивает программист, и командой, которую исполняет процессор.

На практике программисты обнаружили, что они часто повторяют одинаковую `последовательность` инструкций вновь и вновь для того, чтобы делать одно и то же в различных частях программы. Было бы приятно завести имена, представляющие собой каждую из таких обычных последовательностей.

Это пожелание было удовлетворено "макроассемблером", более сложным ассемблером, который мог распознавать не только нормальные инструкции, но также специальные имена ("макро"). Для каждого из них макроассемблер транслирует пять или десять машинных команд, представленных этим именем, так, как будто программист написал их все полностью.

АБСТРАКТНОСТЬ.

Важным достижением было изобретение "языка высокого уровня". Это опять была программа-переводчик, но более мощная. Высокоуровневые языки делают возможным для программистов записывать выражения вида:

X = Y(456/A) - 2

которые сильно похожи на алгебраические. Благодаря языкам высокого уровня инженеры, а не только странноватые бит-жокеи, смогли начать писать программы. Бейсик и Фортран – примеры высокоуровневых языков.

Очевидно, что языки высокого уровня мощнее, чем ассемблеры, поскольку каждая инструкция может компилироваться в десятки машинных команд. Но что более важно, эти языки уничтожают линейное соответствие между исходным текстом и результирующим
машинным кодом.

Реальные инструкции зависят от каждого "выражения" в исходном тексте, взятом как единое целое. Операторы вроде + и = сами по себе не имеют смысла. Они - просто часть сложной символики, которая зависит от синтаксиса и позиции оператора в тексте.

Это нелинейное, зависящее от синтаксиса соответствие между исходным текстом и реальным (обектным) кодом обычно рассматривается как неоценимый вклад в прогресс методологии программирования. Но, как мы увидим впоследствии, такой подход неизбежно предоставляет больше ограничений, чем свободы.

УПРАВЛЯЕМОСТЬ.



А потом я написал GOTO 500 - и вот я здесь!

Большинство компьютеров используют нечто большее, чем просто список инструкций для своей работы. Они также производят проверку различных условий и затем "скачки" в соответствующие части программы в зависимости от результата. Они также производят многократное "зацикливание" на одних и тех же участках кода, обычно контролируя при этом момент выхода из цикла.

Как ассемблер, так и высокоуровневый язык обеспечивают возможности для переходов и циклов. В ассемблерах мы используем команды типа "jump" ("прыжок"), в некоторых языках высокого уровня пользуемся конструкциями типа "GO TO" ("перейти к"). Когда эти возможности используются в сильной степени, программы начинают становиться похожими на такую же неразбериху, как на рисунке 1-1.

Рис.1-1. Неструктурированный код, использующий инструкции типа "jump" или "GOTO".

ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ПРОВЕРКА УСЛОВИЯ
ПЕРЕХОД
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ПЕРЕХОД
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ
ПРОВЕРКА УСЛОВИЯ
ПЕРЕХОД
ПРОВЕРКА УСЛОВИЯ
ПЕРЕХОД

Этот подход, до сих пор широко представленный в таких языках, как Фортран и Бейсик, создает трудности при написании и трудности при внесении изменений. При такой "кашеобразной" манере написания программ невозможно протестировать отдельный участок кода или найти почему выполняется что-то, что выполняться не должно.

Трудности с кашевидными программами привели к открытию "блок-схем". Это были нарисованные карандашом и ручкой картинки, показывающие "течение" процесса, которые использовались программистом в качестве шпаргалки для понимания создаваемого кода. К несчастью, программист вынужден был осуществлять переход от кода к диаграмме и наоборот вручную. Многие программисты осознали бесполезность старомодных диаграмм.

МОДУЛЬНОСТЬ.

Существенное движение вперед произошло с внедрением "структурированного программирования", методологии, основанной на том, что, как показал опыт, большие задачи проще решаются, если рассматривать их как совокупность меньших задач [1]. Каждый такой кусочек называется `модулем`. Программы состоят из модулей внутри других модулей.

Структурированное программирование подавляет кашеобразность кода, поскольку процессы переходов прослеживаются только в пределах модуля. Нельзя перепрыгнуть из середины одного модуля в середину другого.

Например, на рис.1-2 показана блок-схема модуля под названием "приготовление завтрака", который состоит из четырех подмодулей. Внутри каждого подмодуля можно найти новый уровень сложности, которую вовсе не нужно показывать на нашем уровне.

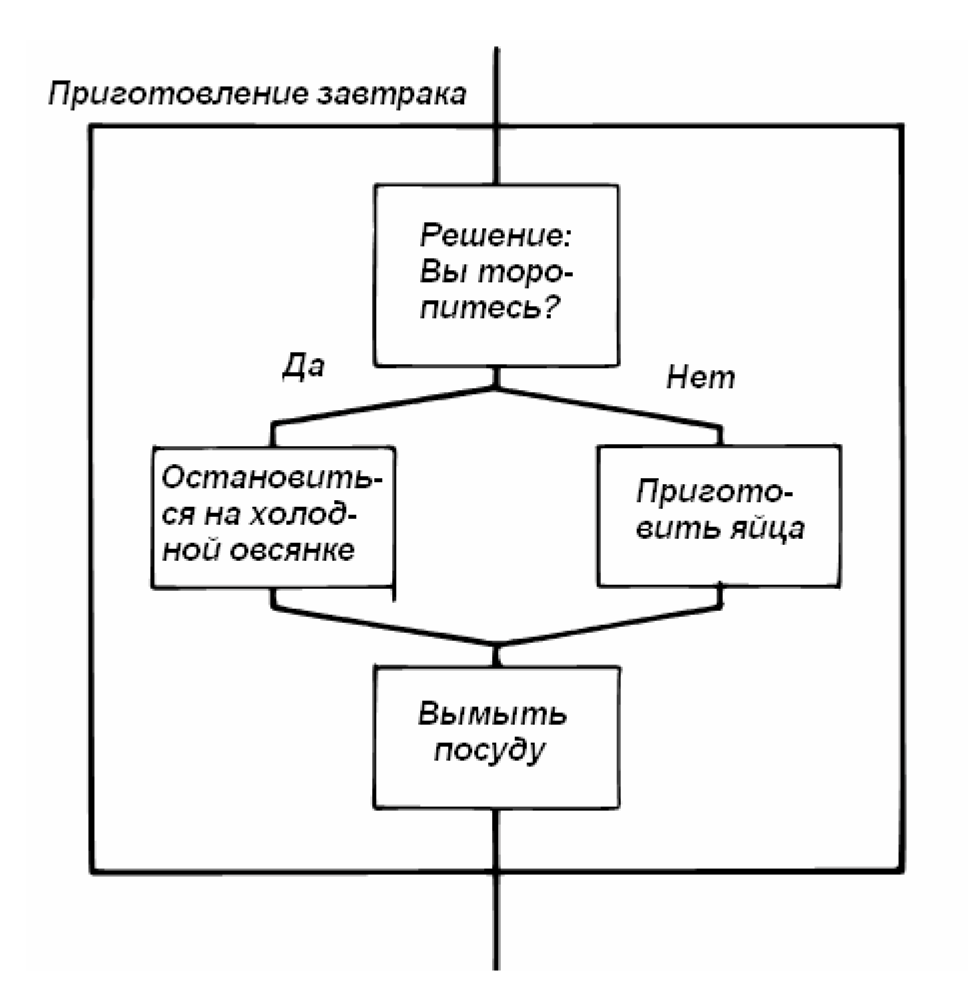


Рис.1-2. Проект структурированной программы.

Решение о переходе внутри нашего модуля принимается при выборе между модулем "холодная овсянка" и модулем "яйца", но линии переходов входят только в наружный модуль.

Структурированное программирование имеет три преимущества:

1. Каждая программа представляется линейной последовательностью содержательных функций, называемых `модулями`. Каждый модуль имеет ровно один вход и ровно один выход.

2. Каждый модуль состоит из одной или нескольких функций, каждая из которых имеет также ровно один вход и ровно один выход и сама может рассматриваться как модуль.

3. Модуль может содержать:

а) операции или другие модули;

б) структуры принятия решений (выражения типа ЕСЛИ ТО);

в) структуры для организации циклов.

Смысл модулей, имеющих "один вход, один выход", состоит в том, что Вы можете вынуть их, изменить их начинку и вставить обратно без развинчивания остальных соединений в программе. Это означает, что Вы можете попробовать каждый кусок по отдельности. Такое возможно когда Вы точно знаете что имеется при входе в модуль и что наблюдается после выхода из него.

В "приготовлении завтрака" Вы либо останавливаетесь на овсянке, либо варите яйца, но не одновременно. А потом Вы обязательно моете посуду. (Насколько мне

известно, некоторые программисты обходят этот последний модуль, переезжая на новую квартиру каждые три месяца.)

Структурированное программирование было изначально задумано как подход к проектированию. Модули были воображаемыми обектами, которые существовали в голове программиста или разработчика и не были частями реального кода. Когда техника структурированного программирования применяется к неструктурированным языкам типа Бейсика, результат получается похожим на то, что показано на рис.1-3.

Рис.1-3. Структурированное программирование на неструктурированном языке.

```
10   ИНСТРУКЦИЯ
20   ИНСТРУКЦИЯ                           Решить - спешим?
30   ЕСЛИ Н=ВЕРНО ТО ПЕРЕЙТИ К 80         если да, то на 80
40   ИНСТРУКЦИЯ
50   ИНСТРУКЦИЯ                           Варка яиц
60   ИНСТРУКЦИЯ
70   ПЕРЕЙТИ К 110                        на 110
80   ИНСТРУКЦИЯ
90   ИНСТРУКЦИЯ                           Приготовление овсянки
100  ИНСТРУКЦИЯ
110  ИНСТРУКЦИЯ                           Мытье посуды
120  ИНСТРУКЦИЯ
```

УДОБСТВО НАПИСАНИЯ.

Следующий шаг вперед, вдохновленный использованием стуктурированных программ - структурированные языки программирования. Они содержат специальные операторы для управления процессом в составе своих наборов команд, что делает возможным написание программ, имеющих более модульный вид.

Таким языком является Паскаль, изобретенный Никлаусом Виртом для обучения студентов принципам структурированного программирования.

На рисунке 1-4 показано, как этот тип языка позволяет переписать программу "приготовление завтрака".

Рис.1-4. Использование структурированного языка.

```
ИНСТРУКЦИЯ
ИНСТРУКЦИЯ              Решение - спешим?
    ЕСЛИ ДА, ТО
        ИНСТРУКЦИЯ
        ИНСТРУКЦИЯ              Варка яиц
        ИНСТРУКЦИЯ
    ИНАЧЕ
        ИНСТРУКЦИЯ
        ИНСТРУКЦИЯ              Приготовление овсянки
        ИНСТРУКЦИЯ
ДАЛЬШЕ
ИНСТРУКЦИЯ              Мытье посуды
ИНСТРУКЦИЯ
```

Языки структурированного программирования имеют управляющие структурные операторы типа ЕСЛИ и ТО для подчеркивания модульности в организации передачи управления. Как Вы можете заметить, отступы в тексте программы важны для ее читабельности, хотя все равно все инструкции внутри модуля написаны полностью вместо замены модуля его именем (например, "приготовление-овсянки"). Законченная программа может занимать десять страниц, с оператором ИНАЧЕ на странице пять.

РАЗРАБОТКА "С ВЕРШИНЫ".

Как приступать к разработке подобных модулей? Методология, называемая "разработкой сверху-вниз", утверждает, что модули следует строить, начиная с самого общего, главного и далее прорабатывать до уровня самых мелких модулей.

Последователи такого подхода могут засвидетельствовать возникновение позорно огромных потерь времени в результате ошибок в планировании. На горьком опыте они познали, что попытки корректировать программу после того, как она была написана - такая практика известна как "наложение заплат" - подобны попытке запереть двери конюшни после того, как лошадь уже понесла.

Поэтому как контрмеру они предлагают следующее официальное правило программирования сверху-вниз:

Не писать ни строчки текста до тех пор, пока
план не проработан до мельчайших деталей.

Вследствие таких трудностей при внесении изменений в однажды написанные программы, все упущения в проекте должны быть устранены на стадии предварительного планирования. В противном случае могут быть затрачены человеко-года усилий по написанию кода, который потом нельзя использовать.

ПОДПРОГРАММЫ.

Мы обсуждали "модули" только как абстрактные объекты. Но любые высокоуровневые языки имеют аппарат, позволяющий кодировать модули проекта как модули реального кода – отдельные куски, которым можно дать имена и "вызывать" из других кусков кода. Эти куски называются подпрограммами, процедурами или функциями, в зависимости от языка программирования и способа реализации.

Предположим, мы написали "приготовление овсянки" в виде подпрограммы. Это могло бы выглядеть как-нибудь вроде:

```
процедура приготовление-овсянки
        взять чистую тарелку
        открыть коробку с овсянкой
        насыпать овсянки
        открыть молоко
        налить молоко
        взять ложку
конец
```

```
10   LET  A = B
20   FOR I = 1 TO 17
30   PRINT  I, BP
40   NEXT [redacted]REM  GET [redacted]
50   PR[redacted](A+B)*([redacted]=QB
60   IF [redacted]>MID$(A[redacted]EN(ZB$))
70   GOTO [redacted]
80   IF Q $([redacted]HAR, 9, 9)= 2 THEN 150
90   NEXT BP; REM: ADJUST
100  GOSUB  160
110  GOTO 10
120  PRINT "[redacted]THE DIRECTORY"
130  INPUT "[redacted]NG" ZP$
140  [redacted]
150
160  PQ= A[redacted]C(D+E)
170  PRINT "INV[redacted] INPU[redacted]
180  FOR Z [redacted]
190  IF ZJ[redacted]
200  A = B;[redacted]
210  NEXT [redacted]: GET MORE
220  RETURN
```

Software patches are ugly and conceal structural weaknesses.

Мы можем также написать и "варку-яиц", и "мытье-посуды" в виде подпрограмм. В этом случае можно определить "приготовление-завтрака" как простую программу, которая вызывает эти подпрограммы:

```
процедура приготовление-завтрака
        переменная С: булевская (означает спешку)
        `проверить наличие спешки`
        если С=истина, то
            вызвать приготовление-овсянки
        иначе
            вызвать варку-яиц
        конец
        вызвать мытье-посуды
конец
```

Фраза "вызвать приготовление-овсянки" заставляет выполниться подпрограмму с таким именем. Когда выполнение заканчивается, управление возвращается назад в вызывающую программу в точку, следующую за вызовом. Подпрограммы повинуются законам структурированного программирования.

Как можно видеть, эффект при использовании подпрограмм такой же, как если бы тело этих подпрограмм присутствовало бы в вызывающем модуле. Но, в отличие от кода, производимого макроассемблером, подпрограмма может быть скомпилирована где угодно в памяти, после чего на нее можно просто ссылаться. Не обязательно компилировать ее внутри реального кода главной программы (см. рис. 1-5).

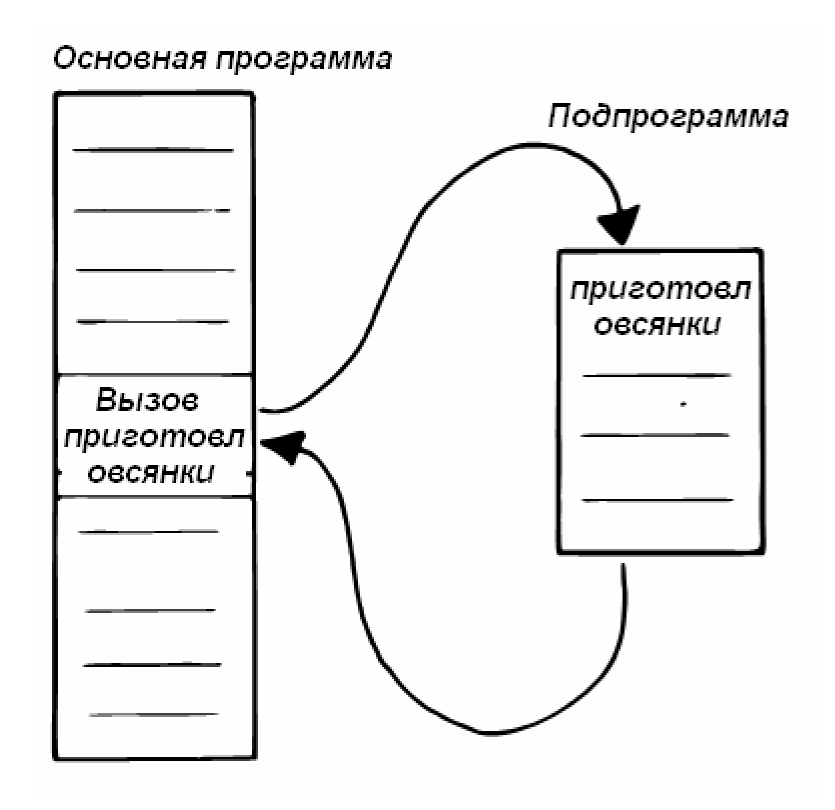


Рис.1-5. Главная программа и подпрограмма в памяти.

Годами ученые-компьютерщики совершенствовались в искусстве использования многочисленных маленьких подпрограмм в сильно разветвленных, протяженных программах. Они могут быть написаны и отлажены независимо друг от друга. Это

облегчает повторное использование ранее написанных программ, и так легче распределять части работы между различными программистами. Короткие куски проще продумывать и проверять их правильность.

Когда подпрограммы компилируются в отдельных частях памяти и вызываются ссылками на них, можно использовать одну и ту же подпрограмму много раз без излишнего расходования места на повторы кода подпрограммы. Так разумное использование подпрограмм может уменьшать размеры кода.

К сожалению, при этом имеется проигрыш в скорости исполнения. Проблему создает необходимость сохранения содержимого регистров перед переходом на подпрограмму и их восстановления при возвращении оттуда. Еще больше времени требуют невидимые, но существенные участки кода, необходимые для передачи параметров в и из подпрограммы.

Существенен также сам способ вызова и передачи параметров подпрограмме. Для автономного тестирования подпрограммы приходится писать специальные тестовые программы для ее вызова.

По этой причине ученые рекомендуют умеренное использование подпрограмм. На практике они обычно получаются довольно большими, от половины до целой страницы текста в длину.

ПОСТЕПЕННАЯ ДЕТАЛИЗАЦИЯ.

Один из подходов, существенно опирающихся на использование подпрограмм, называется "постепенной детализацией" [2]. Идея состоит в том, что Вы начинаете с написания скелетной версии программы, использующей естественные имена процедур и структур данных. Затем Вы пишете версии для каждой из именованных процедур. Процесс продолжается в сторону большего уровня детализации до тех пор, пока процедуры не смогут быть выражены непосредственно на компьютерном языке.

При каждом шаге программист должен принимать решения об используемых алгоритмах и о типах структур данных, которые обрабатываются этими алгоритмами. Такие решения должны приниматься параллельно.

Если выбранный путь оказался непригодным, программист должен набраться мужества вернуться назад насколько требуется и заново проделать работу.

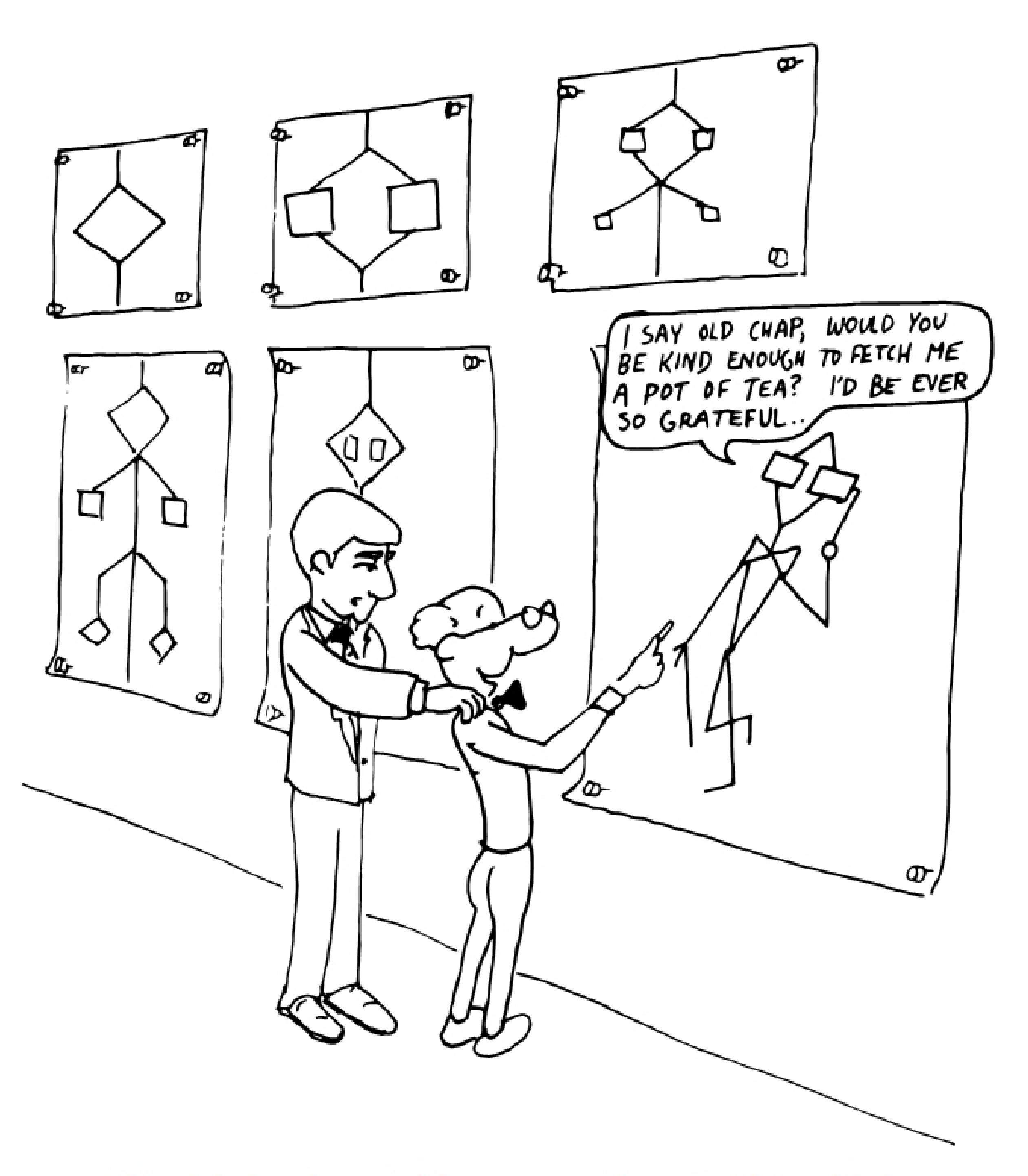
Обратите внимание, что при постепенной детализации Вы не можете реально запустить какую-либо часть программы до тех пор, пока не будут написаны ее компоненты самого нижнего уровня. Обычно это значит, что программу нельзя проверить до тех пор, пока она не будет полностью закончена.

Заметьте также: постепенная детализация заставляет Вас прорабатывать все детали структур управления на данном уровне перед переходом на следующий уровень вниз.

СТРУКТУРИРОВАННАЯ РАЗРАБОТКА.

К середине конца 70-х компьютерная индустрия уже перепробовала все описанные нами концепции и все равно оставалась несчастной. Цена поддержки программного обеспечения - сохранения его функциональности перед лицом возможных изменений - выливалась более чем в половину его общей стоимости, в некоторых случаях доходя до девяноста процентов!

Все соглашались, что эти издержки можно обычно отнести к неполному анализу программ или плохому замыслу разработчиков. Однако было очевидно, что что-то не так с самим структурированным программированием. Когда проекты появлялись с



Tobias, I think you've carried the successive refinement of that module far enough.

опозданием, некомплектными или некачественными, разработчики жаловались на то, что все предвидеть невозможно.

Ученые мужи прилагали все больше усилий к проекту. "В следующий раз мы все продумаем лучше".

К этому времени возникла новая философия, описанная в статье, названной "Структурированная разработка" [3]. Один из ее принципов приводится ниже:

Простота - главный показатель, по которому рекомендуется выбирать среди альтернативных проектов для обеспечения снижения времени на отладку и модификацию. Уровень простоты может быть улучшен за счет разбиения системы на отдельные куски

так, чтобы каждый из них мог рассматриваться, применяться, утверждаться и изменяться с минимальным влиянием на или изменениями в других частях системы.

Разбиение задачи на простые модули должно было облегчить написание, изменение и понимание программ.

Но на какой основе производить разбиение данного конкретного модуля? Статья "Структурированная разработка" выделяет три фактора при проектирования модулей.

ФУНКЦИОНАЛЬНАЯ МОЩНОСТЬ.

Первый фактор - нечто, называемое "функциональной мощностью" - выражает единообразие назначения всего внутри модуля. Если все эти выражения в совокупности могут быть представлены как выполняющие единую задачу, то они являются функционально ограниченными (связными).

В общем случае можно сказать, являются ли выражения в модуле функционально ограниченными, отвечая на следующие вопросы: первый: можно ли описать их назначение одной фразой? Если нет, то модуль, скорее всего, не ограничен функционально. Далее дать ответ на следующие четыре вопроса:

1. Является ли описание составным предложением?
2. Встречаются ли в нем слова - описатели времени, такие, как "сначала", "затем", "потом" и т.д.?
3. Используется ли после глагола существительное общего или неспециального назначения?
4. Есть ли в нем слова типа "инициализировать", предполагающие выполнение множества различных функций одновременно?

Если Вы ответите "да" на один из этих вопросов, то перед Вами некоторое менее связное построение, нежели функциональный модуль. Слабые формы связи:

`Совпадающая связность` (выражения встречаются несколько раз в одном модуле)

`Логическая связность` (в модуле содержится несколько родственных функций и необходим флаг или параметр для решения о том, какую конкретно выполнять)

`Связность по времени` (имеется группа выражений, исполняющихся одновременно, например, инициализация, но не имеющих иной связи)

`Коммуникационная связность` (в модуле содержится группа выражений, работающих с одним и тем же набором данных)

`Последовательная связность` (когда результат одного выражения служит входными данными для следующего)

Наш модуль "приготовление-овсянки" демонстрирует функциональную связность, поскольку его можно представить как единое целое, несмотря даже на то, что он состоит из нескольких подчиненных задач.

СЦЕПЛЕНИЕ.

Второй догмат структурированной разработки говорит о "сцеплении", меры того, как модули влияют на поведение других модулей. Сильное сцепление считается плохим тоном. В наихудшем случае один модуль изменяет код внутри другого модуля. Опасна даже передача управляющих флагов другим модулям для управления их функциями.

Приемлемой формой сцепления является связь через данные, которая предполагает передачу данных (не управляющей информации) из одного модуля в другой. Даже в этом случае системы гораздо легче строить, если интерфейсы передачи данных между модулями по возможности упрощены.

Когда данные могут быть доступны со стороны многих модулей (например, глобальные переменные), говорят о сильном сцеплении между модулями. Если программисту нужно изменить один из них, велика опасность того, что другие продемонстрируют "побочные эффекты".

Самый надежный способ сцепления данных - это передача локальных переменных в качестве переметров от одного модуля к другому. В результате вызывающий модуль говорит подчиненному:

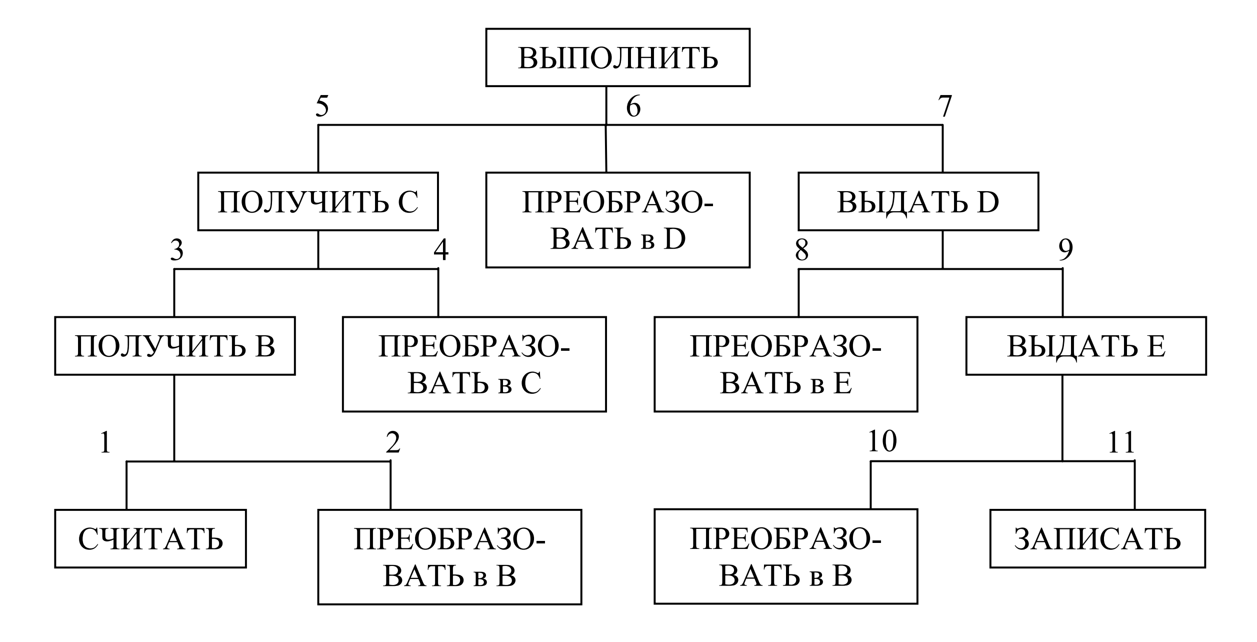
"Я хочу, чтобы ты использовал данные, которые я загрузил в переменные с именами X и Y, а ответ от тебя я ожидаю в переменной по имени Z. Никто другой больше не использует эти переменные".

Как мы уже говорили, обычные языки, поддерживающие подпрограммы, имеют тщательно проработанные методы передачи аргументов от одного модуля к другому.

ИЕРАРХИЧЕСКОЕ ПРОЕКТИРОВАНИЕ ПО ПРИНЦИПУ "ВХОД-ОБРАБОТКА-ВЫХОД".

Третья составляющая структурированной разработки касается процесса проектирования. Разработчикам рекомендуется использовать подход "сверху-вниз", но уделять при этом меньше немедленного внимания на управляющие структуры. "Разработка принятия решений" может подождать до более детализированной проработки модулей. Вместо этого на ранней стадии следует концентрировать усилия на иерархии внутри программы (какие модули вызывают какие модули) и на передаче данных от модуля к модулю.

Рис.1-6. Представление структурной диаграммы, из статьи "Структурированная разработка". (Structured Design, IBM Systems Journal).



	Вход	Выход
1		A
2	A	B
3	B	
4	B	C
5	C	
6	C	D
7	D	
8	D	E
9	E	
10	E	F
11	F	

Для того, чтобы помочь разработчикам думать в этом направлении, было предложено графическое представление, названное "структурными диаграммами". (Несколько измененная форма так называемых "иерархических диаграмм"). Эти диаграммы состоят из двух частей: иерархической схемы и таблицы входов-выходов.

На рис. 1-6 показаны эти две части. Главная программа, названная "выполнить", состоит из трех подчиненных модулей, которые, в свою очередь, вызывают другие модули, изображенные под ними. Как видно, при проектировании увеличивается внимание к преобразованию входных данных в выходные.

Числа в иерархической диаграмме соответствуют строкам в таблице входов-выходов. В точке 1 (модуль СЧИТАТЬ) выходом является величина А. В точке 2 (модуль ПРЕОБРАЗОВАТЬ-в-В), на вход подается А, выходом является В.

Быть может, наибольшим вкладом такого подхода является осознание того, что решения о передаче управления не должны доминировать в проекте. Как мы убедимся, поток управления – это поверхностный аспект проблемы. Мизерные изменения в исходных требованиях могут существенно изменить структуры управления в программе и потребовать многих лет ее углубленного "перекапывания". Но, если проект программы ориентирован на что-то другое, например, на потоки данных, то изменения в планах не будут столь разрушительными.

УПРЯТЫВАНИЕ ИНФОРМАЦИИ.

В работе [4], опубликованной еще в 1972 году, д-р Дэвид Л. Парнас показал, что критерием для разбиения на модули должны быть не шаги в процессе, а куски информации, которые, возможно, будут меняться. Модули должны использоваться для сокрытия такой информации.

Давайте рассмотрим эту важную идею об "упрятывании информации": предположим, Вы пишете Руководство по делопроизводству для своей компании. Вот его фрагмент:

Отдел продаж принимает заказ
посылает синюю копию в архив
оранжевую копию на склад

Джей подшивает оранжевую копию в красный скоросшиватель на своем столе и производит упаковку.

Все согласны, что эта процедура корректна, и Ваше руководство распространяется для всех в компании.

А потом Джей увольняется, а приходит Мэрилин. Новые копии приказов имеют зеленую и желтую обложки вместо синей и оранжевой. Красный скоросшиватель переполняется и уступает место черному.

Все Ваше руководство становится устаревшим. Вы могли бы избежать устаревания, применяя слово "упаковщик" вместо имени "Джей", словосочетания "архивная копия" и "складская копия" вместо "синей" и "оранжевой" и т.д.

Этот пример иллюстрирует мысль о том, что для сохранения корректности перед лицом возможных изменений произвольные детали должны быть исключены из процедур. Они могут быть при необходимости описаны отдельно. К примеру, каждую неделю или около того отдел кадров может издавать список работников и их должностей, так что каждый при необходимости может узнать имя упаковщика из единого для всех источника. При изменении кадрового состава этот список должен будет меняться.

Такая техника очень важна при написании программного обеспечения. Почему же работающая уже программа должна быть когда-нибудь изменена? По любой из миллиона причин. Вам может понадобиться запустить ее на новом оборудовании, программа должна быть изменена только для того, чтобы приспособиться к этому оборудованию. Ей, может быть, необязательно быть чрезвычайно быстрой или мощной для того, чтобы удовлетворить использующих ее людей. Большинство групп разработчиков обнаруживают, что пишут "семейства" программ, то есть много версий родственных программ для конкретного поля применений, все являющиеся вариантами одной ранней версии.

Для обеспечения принципа упрятывания информации определенные детали программы должны быть сведены в единое место, и каждый полезный кусок информации должен встречаться один раз. Программы, игнорирующие эту максиму, виновны в избыточности. В то время как избыточность аппаратуры (резервные ЭВМ и т.д.) могут сделать систему более безопасной, информационная избыточность наоборот, опасна.

Любой знающий программист скажет Вам, что число, которое предположительно может измениться в будущих версиях программы, должно быть определено как "константа" и встречаться в программах в виде ссылки на свое имя, а не в виде значения. Например, количество колонок, представляющих ширину бумаги в Вашем принтере, должно быть представлено константой. Даже в языках ассемблера имеются конструкции типа "EQU" и метки для связи адресов и битовых масок с именами.

Любой хороший программист также применит принцип упрятывания информации при разработке подпрограмм, стремясь к тому, чтобы каждый модуль знал как можно меньше о содержимом других модулей. Сегодняшние языки программирования, такие, как C, Модула-2 и Эдисон, применяют этот подход в архитектуре своих процедур.

Но Парнас проводит свою идею значительно дальше. Он предлагает распространить ее на алгоритмы и структуры данных. Упрятывание информации - а не структура принятия решений или иерархия вызовов - вот что должно лежать в основе проектирования!

ПОВЕРХНОСТНОСТЬ СТРУКТУРЫ

Парнас предлагает два критерия разбиения:

а) возможное (хотя пока и незапланированное) использование и
б) возможное (хотя и незапланированное) изменение.

Этот новый взгляд на "модуль" отличается от традиционного.
Такой "модуль" - собрание кусочков, обычно очень маленьких, которые вместе прячут информацию о некоторой стороне проблемы.

Двое других авторов описывают ту же идею по-другому, используя выражение "абстрагирование данных" [5]. Они приводят в пример стековую структуру. Стековый "модуль" состоит из процедур для очистки стека, отправки значения в стек, снятия значения оттуда и определения пустоты стека. Этот "многопроцедурный" модуль скрывает информацию о том как оргазизован стек от остальных частей программы. При этом он считается единым модулем потому, что его процедуры взаимозависимы. Нельзя изменить способ загрузки значения в стек без изменения способа снятия его оттуда.

Слово `использовать` играет важную роль в этой концепции. Парнас пишет в своей более поздней статье [6]:

Системы, достигнувшие определенной степени "элегантности" ... сделали это за счет "использования" одними частями системы других ...

Если существует такое иерархическое построение, то каждый уровень представляет собой проверяемый и способный быть использованным подраздел системы ...

Проектирование иерархии "использования" может быть одним из главных направлений приложения сил. Расчленение системы на независимо вызываемые подпрограммы должно производиться параллельно с решениями об "использованиях", поскольку они влияют друг на друга.

Разаработка, при которой модули группируются в соответствии с потоком передачи управления, не готова к быстрым изменениям в проекте. Структура, построенная в соответствии с иерархией управления, является поверхностной.

Готовность к восприятию изменений может обеспечить проект, в котором модули формируются по признаку объединения потенциально изменяемых вещей.

ВЗГЛЯД НАЗАД, ВПЕРЕД И НА ФОРТ (*)

(*) - игра слов: Форт (FORTH) означает "вперед" (англ.) (здесь и далее примечания переводчика).

В этом разделе мы рассмотрим основные свойства языка Форт и сравним их со свойствами традиционных методологий.

Вот пример текста на Форте:

```
: ЗАВТРАК
        СПЕШИМ?  IF   ОВСЯНКА   ELSE   ЯЙЦА   THEN  МЫТЬЕ ;
```

Он по структуре идентичен процедуре "приготовление-завтрака" на стр.8. (Если Вы - новичок в Форте, обратитесь к приложению А за объяснениями.)

Слова СПЕШИМ?, ОВСЯНКА, ЯЙЦА и МЫТЬЕ также заданы (что наиболее вероятно) как определения через двоеточие.

Здесь Форт демонстрирует все положительные качества, изученные нами: мнемонические обозначения, абстракцию, мощность, структурированные операторы передачи управления, сильную функциональную ограниченность, небольшую степень связности и модульность. Но, кроме модульности, подчеркнем еще то, что, быть может, является наиболее важной заслугой Форта:

Мельчайшим атомом программы на Форте является не модуль или подпрограмма или процедура, а "слово".

Далее, отсутствуют понятия подпрограмм, главных программ, утилит или операторов, вызываемых по отдельности. `Все` в Форте есть слова.

Перед тем, как мы изучим важность среды, основанной на словах, давайте остановимся на двух новшествах Форта, которые делают это возможным.

АВТОМАТИЧЕСКИЕ ВЫЗОВЫ.

Во-первых, вызовы производятся автоматически. Нет необходимости писать CALL ОВСЯНКА, достаточно просто ОВСЯНКА. В Форте определение ОВСЯНКА "знает", какого типа словом оно является и какую процедуру надо вызвать для исполнения себя.

Таким образом переменные и константы, системные функции, утилиты, так же, как и определенные пользователем команды или структуры данных, могут быть "вызваны" просто по имени.

АВТОМАТИЧЕСКАЯ ПЕРЕДАЧА ДАННЫХ.

Во-вторых, передача данных происходит сама собой. Механизм, производящий такой эффект - это стек данных Форта. Форт автоматически загружает числа на стек; слова, требующие на входе числа, автоматически снимают их оттуда; слова, выдающие на выходе значения, автоматически кладут их на стек. Слова ПОЛОЖИТЬ-НА-СТЕК и ВЗЯТЬ-СО-СТЕКА в Форте на высоком уровне не существуют.

Таким образом, мы можем написать:

```
: ВЫПОЛНИТЬ
        ПОЛУЧИТЬ-С  ПРЕОБРАЗОВАТЬ-в-D  ВЫДАТЬ-D ;
```

где ПОЛУЧИТЬ-С считывает "С" и оставляет его на стеке, ПРЕОБРАЗОВАТЬ-в-D берет "С" со стека, преобразует и оставляет на стеке как "D". Наконец, ВЫДАТЬ-D берет "D" со стека и записывает его. Форт скрывает акт передачи данных в нашем коде, позволяя сконцентрироваться на функциональных шагах преобразования информации.

Вследствие использования стека для передачи данных слова могут вкладываться в другие слова. Любое слово может положить числа на стек и взять их назад, не нарушая поток данных между вышестоящими словами (разумеется, если оно не забирает или оставляет на стеке какие-нибудь неожиданные значения). Таким образом, стек обеспечивает структурированное, модульное программирование, в то же время предоставляя простой механизм передачи локальных аргументов.

Форт убирает из наших программ детали о том, `как` вызываются слова и `как` передаются данные. Что же остается? Остаются только слова, описывающие нашу задачу.

Имея слова, мы можем полностью воспользоваться рекомендациями Парнаса - разбивать задачу в соответствии с частями, которые могут измениться, и формировать каждый "модуль" из стольких маленьких функций, сколько их потребуется для упрятывания информации об этом модуле. На Форте мы можем написать для этого столь много слов, сколько для этого нужно, независимо от простоты каждого из них.

Строка из типичной для Форта задачи может выглядеть так:

```
20 ПОВЕРНУТЬ ВЛЕВО БАШНЯ-ТАНКА
```

Немногие другие языки могут вдохновить Вас на сочетание подпрограммы с именем ВЛЕВО, просто модификатором, с подпрограммой БАШНЯ-ТАНКА, просто именем части аппаратуры.

Поскольку слово Форта легче запустить, чем подпрограмму (просто по имени, без специального вызова), программа на Форте может быть разбита на большее количество слов, нежели обычная - на подпрограммы.

ПРОГРАММИРОВАНИЕ НА УРОВНЕ КОМПОНЕНТОВ

Наличие большого набора простых слов делает простым использование техники, которую мы назовем "программирование на уровне компонентов". Для пояснения давайте вначале просмотрим те объединения, которые мы неопределенно описали как "части, которые могут быть изменены". В типовой системе почти все может быть подвержено изменениям: устройства ввода/вывода, такие, как терминалы и принтеры, интерфейсы типа микросхем ПЛМ, операционные системы, любые структуры данных или их представление, любые алгоритмы и т.д.

Вопрос: "Как можно минимизировать вклад каждого из подобных изменений? Как определить наименьший набор изменяемых вещей для обеспечения требуемых перемен?"

Ответом является: "наименьший набор взаимовлияющих структур данных и алгоритмов, которые разделяют знание о том, как они в совокупности работают." Мы будем называть такое объединение "компонентом".

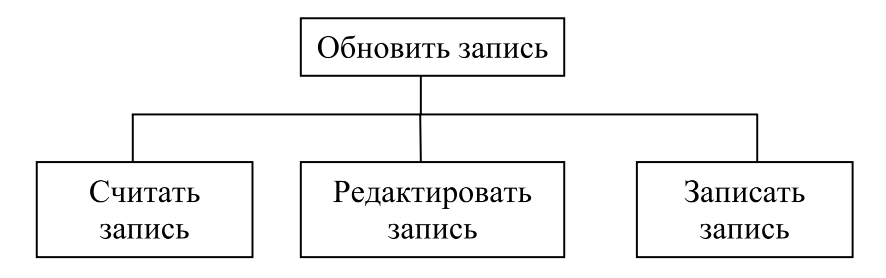
Компонент является ресурсом. Он может быть частью аппаратуры, например, ПЛМ или аппаратным стеком. Или компонент может быть программным ресурсом, таким, как очередь, словарь или программный стек.

Все компоненты включают в себя объекты данных и алгоритмы. Не имеет значения, физический ли это объект данных (такой, как аппаратный регистр) или абстрактный (как вершина стека или поле в базе данных). Безразлично, описан ли алгоритм в машинном коде или проблемно-ориентированными словами типа ОВСЯНКА или ЯЙЦА.

Рисунок 1-7 сопоставляет результаты структурированной разработки с результатами разработки на уровне компонентов. Вместо `модулей` под названиями ЧИТАТЬ-ЗАПИСЬ, РЕДАКТИРОВАТЬ-ЗАПИСЬ и ПИСАТЬ-ЗАПИСЬ мы сосредоточены на `компонентах`, описывающих структуру записей, реализующих систему команд редактора и обеспечивающих процедуры чтения/записи.

Что мы сделали? Мы ввели новый этап в процесс разработки: разбили на компоненты во время `проектирования`, затем описали последовательность, иерархию и линию вход-обработка-выход при `реализации`. Да, это еще один шаг, однако мы получили дополнительное измерение для проведения разреза - не только по слоям, но и `в клеточку`.

Рис.1-7. Структурированная разработка против разработки на уровне компонентов.
Последовательное/иерархическое проектирование:



Разработка по компонентам:

```
: ОБНОВИТЬ-ЗАПИСЬ
     ЗАПИСЬ ЧИТАТЬ   РЕДАКТИРОВАТЬ  ЗАПИСЬ ПИСАТЬ ;
```

Структура записей

Редактор

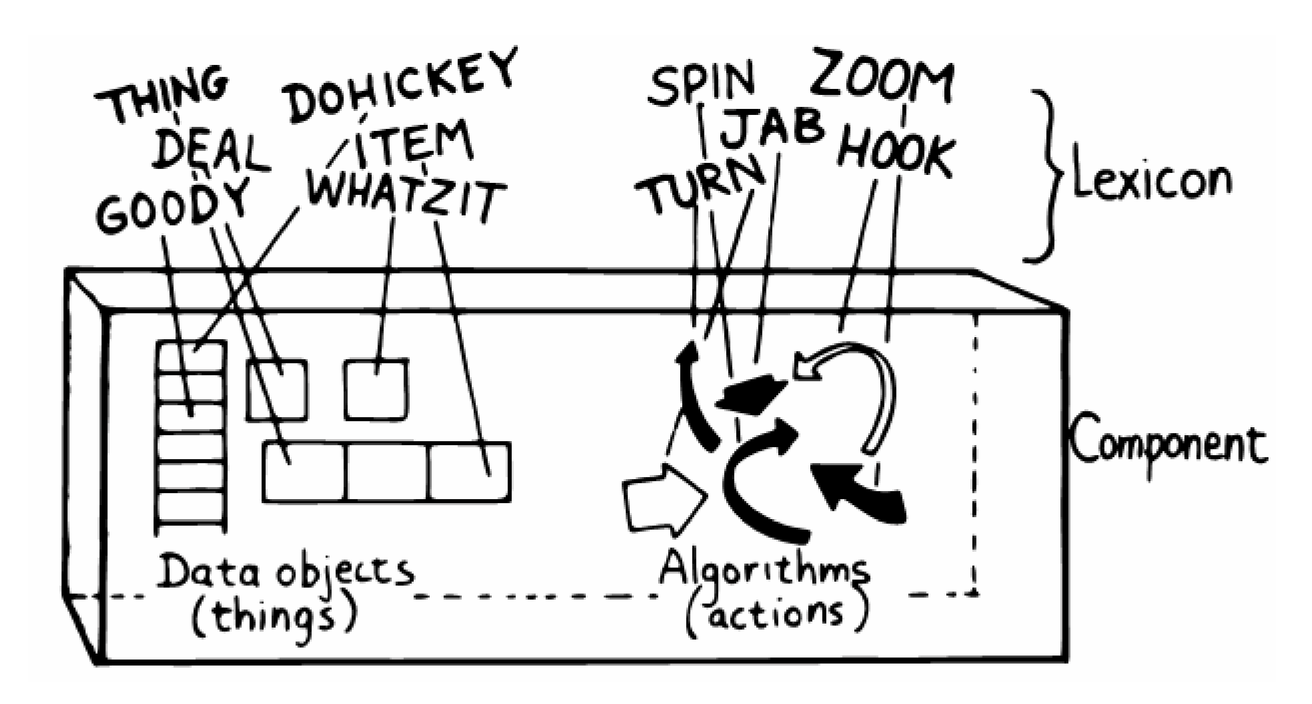
Программы чтения/записи

Представим себе, что после того, как программа была написана, нам потребовалось изменить структуру записи. При последовательном, иерархическом проектировании это затронет все три модуля. При проектировании на уровне компонентов изменение коснется лишь компонента, описывающего структуру записи. Ничто другое, использующее этот компонент, не должно знать о перемене.

В дополнение к перечисленному преимущество такой схемы состоит в том, что программисты в группе могут получить в разработку индивидуальные компоненты с меньшей взаимозависимостью. Принцип программирования компонентов применим к руководству группой так же, как и к разработке программ.

Мы будем называть набор слов для описания компонента "лексиконом". (Одно из значений слова "лексикон" - "набор слов, относящийся к определенному кругу интересов".) Лексикон – это Ваш наружный интерфейс с компонентами (рис. 1-8).

Рис.1-8. Компонент описывается лексиконом.



В данной книге слово "лексикон" относится только к тем словам компонента, которые используются по имени вне этого компонента. Компонент может содержать также определения, написанные исключительно для поддержки видимого снаружи лексикона. Мы будет называть вспомогательные определения "внутренними" словами.

Лексикон дает логические эквиваленты объектам данных и алгоритмам в форме имен. Лексикон вуалирует структуры данных и алгоритмы компонентов - "как оно работает". Он представляет миру только "концептуальную модель" компонента, описанную простыми словами - "что оно делает".

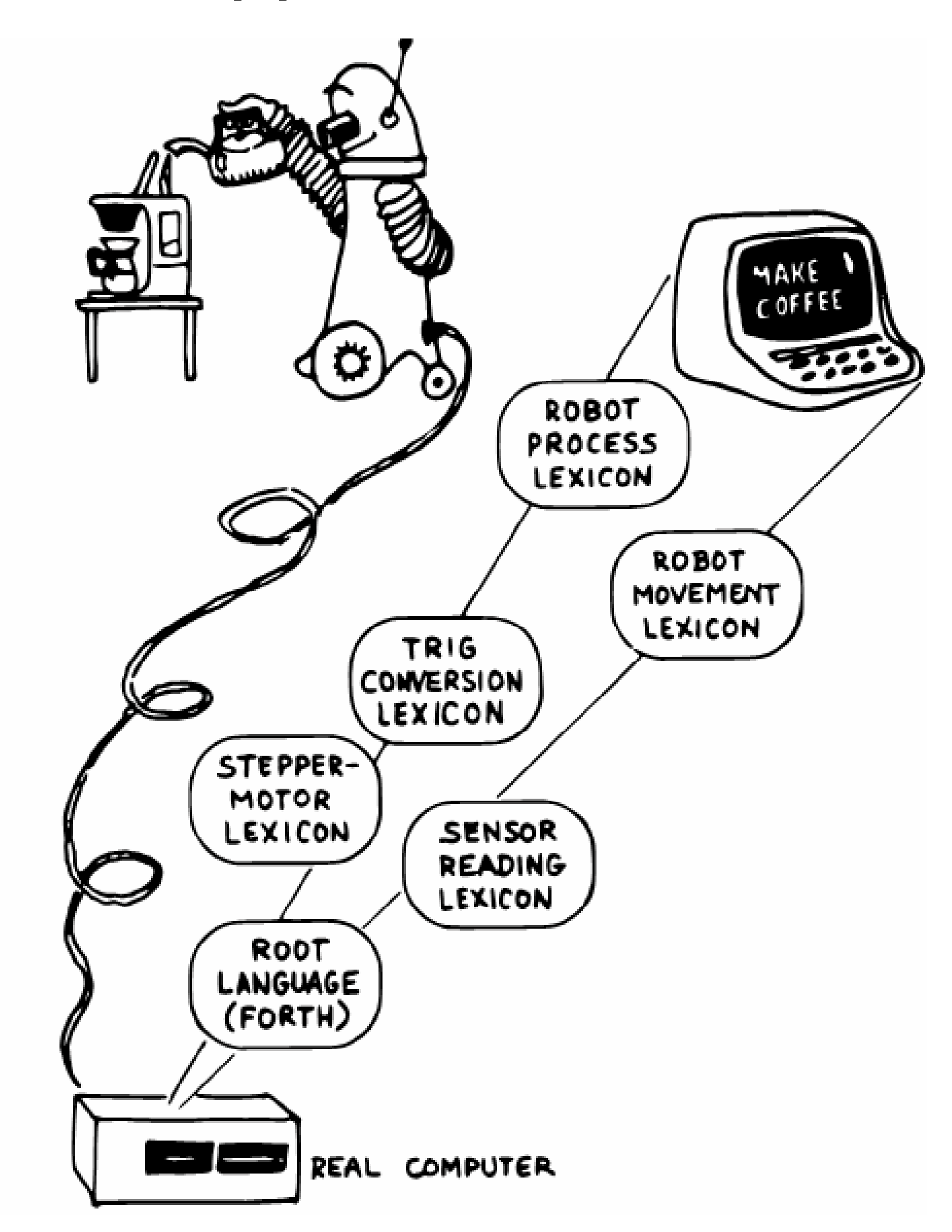
Эти слова затем становятся языком для описания структур данных и алгоритмов компонентов, написанных на более высоком уровне. "Что" для одного компонента становится "как" для высших компонентов.

Написанная на Форте задача состоит только из компонентов. Рисунок 1-9 показывает, как может быть разбита робототехническая задача.

Можно даже сказать, что лексикон - это специализированный компилятор, написанный специально для поддержки кода высокоуровневой программы наиболее эффективным и надежным способом.

Между прочим, сам по себе Форт не поддерживает компоненты. Ему это не нужно. Компоненты - это продукты разбиения программы ее проектировщиком. (В то же время Форт имеет "блоки" - небольшие порции массовой памяти для хранения исходных текстов. Компонент обычно может быть написан в пределах одного или двух экранов Форта.)

Рис.1-9. Полная программа состоит из компонентов.



Важно понять, что лексикон может использоваться любым или всеми компонентами высших уровней. Ни один нормальный компонент `не` прячет свои компоненты поддержки, как это часто случается при по-слойном подходе к разработке. Вместо этого каждый лексикон волен использовать все команды, определенные до него.

Команда для движения робота опирается на корневой язык, со всеми его переменными, константами, операторами работы со стеком, математикой и др. так же сильно, как и на любой другой компонент.

Важным результатом такого подхода является то, что полная задача использует единый синтаксис, который легко выучить и соблюдать. Именно поэтому я использую слово "лексикон", а не "язык". У языков уникальные синтаксисы.

Доступность команд также значительно облегчает процесс тестирования и отладки. Интерактивность Форта позволяет программисту набирать и тестировать примитивные команды типа

ПРАВЫЙ ПЛЕЧО 20 ПОВЕРНУТЬ

"снаружи" так же просто, как и более мощные типа

ЛЕВЫЙ КОФЕЙНИК

В то же время программист может (при желании) намеренно запретить доступ конечного пользователя к любым командам, включая сам Форт, после того, как программа закончена.

Новая Форт-методология проясняется. Программирование на Форте состоит в расширении корневого языка в сторону приложения, определении новых команд, которые прямо описывают проблему.

Языки программирования, созданные специально для опеределенных применений, таких как робототехника, производственный контроль, статистика и т.д., известны как "проблемно-ориентированные". Форт - это программные средства для `создания` проблемно-ориентированных языков. (Последняя фраза - быть может, самое краткое из возможных описаний Форта.)

На самом деле Вам не стоит писать каких-либо серьезных задач на Форте; как язык, он просто недостаточно мощен. Вам `следует` писать на Форте свои собственные языки (лексиконы) для моделирования Вашего понимания проблемы, на которых Вы можете элегантно описать ее решение.

ОТ КОГО ПРЯТАТЬ?

Поскольку современные языки программирования дают несколько иное толкование выражения "упрятывание информации", нам придется внести ясность. От чего, от кого мы прячем информацию?

Новейшие традиционные языки (такие как Модула-2) напрягают свои силы для обеспечения упрятывания в модуле информации о его внутренних алгоритмах и структурах данных от других модулей.

Целью является достижение независимости модуля (минимальной связности). Создается впечатление, что модули стараются атаковать друг друга как враждебные антитела. Или по-другому, что злобные банды модулей-мародеров выходят на большую дорогу грабить драгоценное семейство структур данных.

Это `не` то, чем озабочены мы. Мы понимаем упрятывание информации просто как средство минимизации эффектов от возможного изменения проекта методом локализации тех вещей, которые могут измениться, внутрь каждого компонента.

Форт-программисты обычно предпочитают держать свою программу под личным контролем и не использовать технику физического упрятывания структур данных. (Несмотря на это, великолепно простой способ - всего в три строки исходного текста – добавлении я к Форту Модула-подобных модулей был предложен Дьювеем Валь Шорром [7].)

УПРЯТЫВАНИЕ КОНСТРУКЦИИ СТРУКТУР ДАННЫХ

Мы уже отмечали две особенности Форта, обеспечивающие использование описанной методологии - автоматические вызовы и автоматическую передачу данных. Третья особенность позволяет описывать структуры данных внутри компонента в терминах предварительно описанных компонентов. Эта особенность – прямой доступ к памяти.

Предположим, что мы определяем переменную ЯБЛОКИ:

```
VARIABLE ЯБЛОКИ
```

Мы можем записать число в эту переменную для указания того,
сколько яблок имеется в текущий момент:

```
20 ЯБЛОКИ !
```

Мы можем распечатать содержимое переменной:

```
ЯБЛОКИ ? 20 ok
```

Мы можем увеличить ее содержимое на единицу:

```
1 ЯБЛОКИ +!
```

(Новичок может изучить механизм работы этих фраз по приложению А.)

Слово ЯБЛОКИ имеет единственную функцию: положить на стек `адрес` в памяти, где хранится количество яблок. О количестве можно думать как о "вещи", в то время как о словах, устанавливающих количество, считывающих или увеличивающих его - как о "действиях".

Форт удобно отделяет "вещи" от "действий", поскольку разрешает передачу адресов через стек и имеет команды "разыменования" и "загрузки".

Мы обсуждали важность проектирования по признаку того, что может измениться. Предположим, мы написали множество кода, использующего переменную ЯБЛОКИ. И теперь, в одиннадцатом часу, обнаруживаем, что необходимо отслеживать два различных типа яблок - красных и зеленых!

Не стоит опускать руки, лучше вспомнить функцию слова ЯБЛОКИ: давать адрес. Если нам нужно два различных количества, ЯБЛОКИ могут давать два различных адреса, в зависимости от того, о каком типе яблок мы говорим. Так мы можем определить более сложную версию слова ЯБЛОКИ, как показано ниже:

```
VARIABLE ЦВЕТ  ( указатель на текущую переменную)
VARIABLE КРАСНЫЕ  ( количество красных яблок)
VARIABLE ЗЕЛЕНЫЕ  ( количество зеленых яблок)
```

```
: КРАСНЫЙ ( тип яблок - красные)  КРАСНЫЕ ЦВЕТ ! ;
: ЗЕЛЕНЫЙ ( тип яблок - зеленые)  ЗЕЛЕНЫЕ ЦВЕТ ! ;
: ЯБЛОКИ  ( -- адр текущей яблочной переменной)  ЦВЕТ @ ;
```

Рис.1-10. Смена косвенного указателя.

Мы переопределили ЯБЛОКИ. Теперь они дают содержимое переменной по имени ЦВЕТ. ЦВЕТ - указатель: либо на переменную КРАСНЫЕ, либо на переменную ЗЕЛЕНЫЕ. Последние две переменных и являются действительными хранилищами для количеств.

Если мы сначала говорим КРАСНЫЙ, то можно использовать ЯБЛОКИ по отношению к красным яблокам, если говорим ЗЕЛЕНЫЙ, то - по отношению к зеленым (рис. 1-10).

Нам не понадобилось изменять синтаксис чего-либо в наработанном коде, использующем ЯБЛОКИ. Мы так же говорим

```
20 ЯБЛОКИ !
```

и

```
1 ЯБЛОКИ +!
```

Взгляните опять на то, что мы сделали. Мы изменили описание слова ЯБЛОКИ с описания переменной на определение через двоеточие, никак не повлияв на метод его использования. Форт позволяет нам скрыть детали того, как определено слово ЯБЛОКИ, от использующего его кода. То, что представляется "вещью" с точки зрения старых определений, в действительности является "действием" (определением через двоеточие) внутри компонента.

Форт вдохновляет на использование абстрактных типов данных, позволяя определять структуры данных в терминах компонентов нижнего уровня. Только Форт, исключающий вызовы (вида CALL) в процедурах, позволяющий передавать адреса и данные через стек и предоставляющий прямой доступ к памяти с помощью слов @ и ! может предложить такой уровень упрятывания информации.

Форт мало заботится о том, является ли что-либо структурой данных или алгоритмом. Это дает нам, программистам, невероятную свободу в создании тех частей речи, которые нам нужны для описания наших задач.

Я стараюсь думать о каждом слове, возвращающем адрес (например, ЯБЛОКИ) как о "существительном" независимо от способа, которым оно определено. Слово, производящее очевидное действие - это "глагол".

Такие слова, как КРАСНЫЙ и ЗЕЛЕНЫЙ в нашем примере, могут быть названы только "прилагательными", поскольку они изменяют функцию слова ЯБЛОКИ. Фраза

КРАСНЫЙ ЯБЛОКИ ?

отличается от фразы

ЗЕЛЕНЫЙ ЯБЛОКИ ?

Слова Форта могут служить также наречиями и предлогами. Мало смысла в том, чтобы определить, какой частью речи является конкретное слово, поскольку Форту в любом случае все равно. Нам нужно лишь порадоваться легкости описания задачи в естественных выражениях.

НО ВЫСОКОУРОВНЕВЫЙ ЛИ ЭТО ЯЗЫК?

В нашем кратком историческом обзоре было замечено, что традиционные языки высокого уровня оторвались от ассемблерных языков, устранив не только соответствие `один в один` между командами и машинными операциями, но и соответствие `линейное`. Очевидно, Форт имеет первое отличие, но что касается второго, то порядок слов, используемых в определении, совпадает с порядком, в котором эти команды компилируются.

Выводит ли это Форт из рядов языков высокого уровня? Перед тем, как ответить, давайте исследуем преимущества Форт-подхода.

Вот что по этому поводу есть сказать у изобретателя Форта Чарльза Мура:

Вы определяете каждое слово так, что ЭВМ знает его значение. Способ, которым она это знает, состоит в том, что при вызове исполняется некоторый последовательный код. Компьютер предпринимает действия сразу по каждому слову. Он не отправляет слово на хранение и не держит его в уме на будущее.

В философском смысле, я думаю, это означает, что машина "понимает" слово. Она понимает слово DUP, быть может, лучше вашего, поскольку в ее мозгах никогда не возникает сомнение по поводу того, что DUP означает.

Связь между словами, имеющими смысл для Вас и имеющими смысл для компьютера, глубока. ЭВМ становится средством для связи между человеческим существом и концепцией.

Одним из преимуществ соответствия между исходным текстом и машинным кодом является огромное упрощение компилятора и интерпретатора. Такое упрощение улучшает работу многих частей системы, как это будет видно из дальнейшего.

С точки зрения методологии программирования, преимущество Форт-подхода состоит в том, что `новые` слова и `новые` синтаксисы могут легко добавляться. Нельзя говорить, что Форт `ищет` слова - он находит слова и исполняет их. Если Вы добавляете новые слова, Форт с тем же успехом найдет и исполнит их. Нет различия между существующими словами и теми, которые добавили Вы.



Две точки зрения

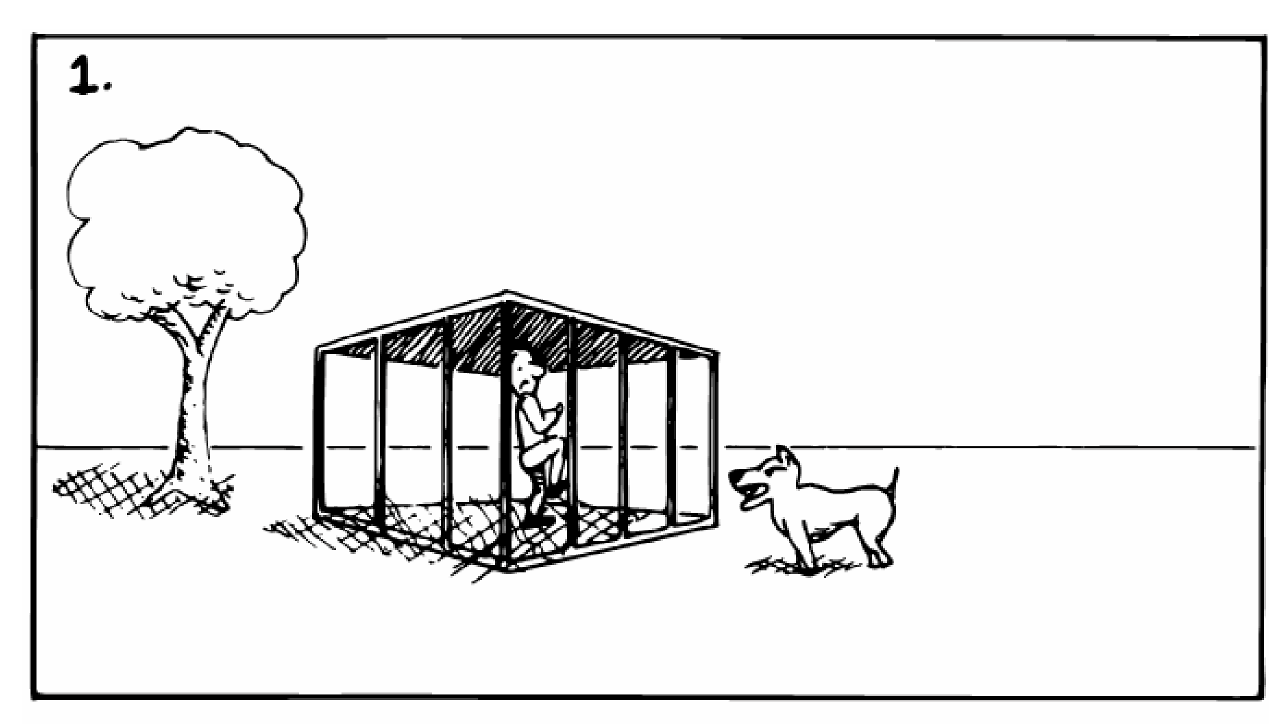
Более того, такая "расширяемость" подходит к любым типам слов, а не только к словам-действиям. К примеру, Форт позволяет Вам добавлять новые `компилирующие` слова - такие, как IF и THEN, которые обеспечивают структурированный поток управления.

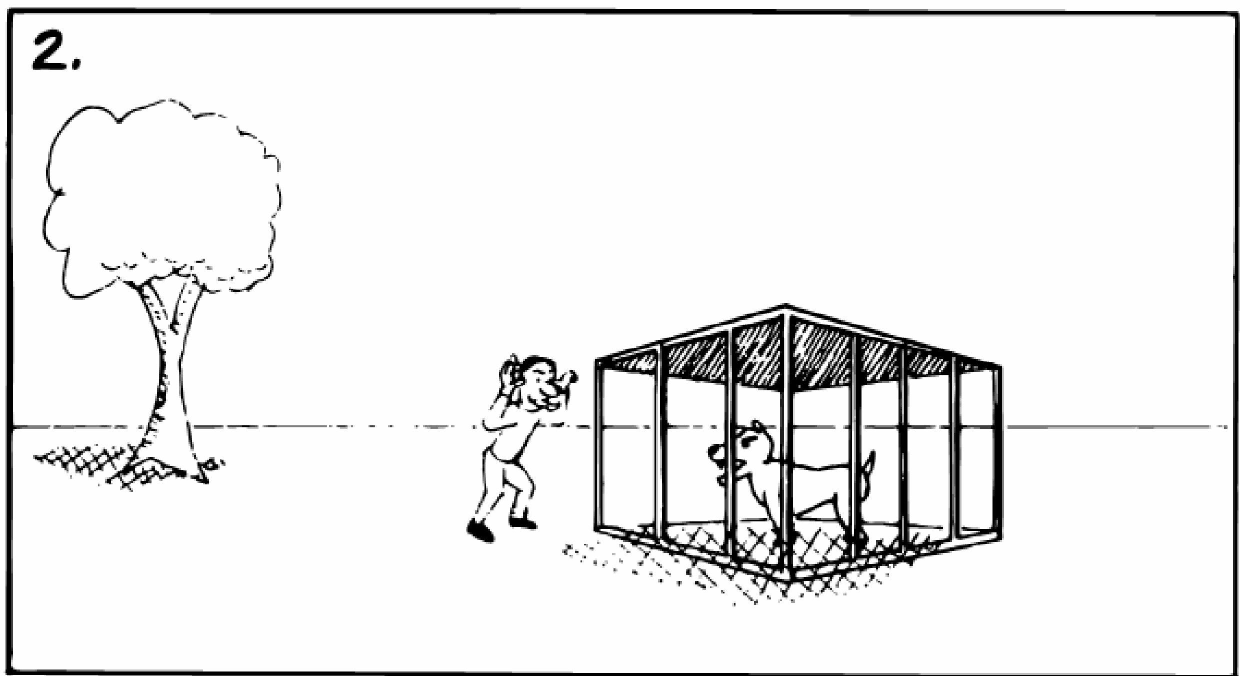
Вам также не составляет труда добавить оператор выбора по множеству вариантов или циклическую структуру со множеством выходов, если они Вам понадобятся или, что тоже важно, убрать их, если они не нужны.

В противоположность этому любой язык, в котором для понимания выражения важен порядок слов, должен "знать" все слова и все их допустимые комбинации. Шансы на то, что в них предусмотрены все удобные для Вас комбинации, малы. Язык таков, каким его создали, Вам нельзя расширить его знания.

Исследователи в лабораториях называют гибкость и расширяемость Форта среди его наиболее важных преимуществ для их работы. Можно разрабатывать лексиконы для сокрытия информации об огромном разнообразии тестового оборудования, присоединенного к компьютеру. Когда такая работа проделана более опытным программистом, исследователи могут использовать свой "программный инструментарий" маленьких слов для написания простых экспериментальных программ. При появлении нового оборудования добавляются новые лексиконы.

Марк Бернстейн описал проблему использования готовой целевой библиотеки процедур в лаборатории [8]: "Компьютер, а не пользователь, доминирует в эксперименте". Но, как он пишет, с помощью Форта "компьютер действительно подвигает ученых на изменение, исправление и улучшение программного обеспечения, для экспериментирования и изучения особенностей своего оборудования. Инициатива снова становится прерогативой исследователя."





Два решения проблемы безопасности

Тех, кто упорствует в том, что Форт нельзя назвать языком высокого уровня, последний снабжает дополнительными аргументами. В то время, когда мощная проверка на синтаксис и типы данных становится одним из главных направлений в современных языках программирования, Форт вообще почти не производит синтаксический анализ. Предоставляя ту гибкость и свободу, которую мы описали, он не может указать Вам, что Вы собирались написать КРАСНЫЙ ЯБЛОКИ вместо ЯБЛОКИ КРАСНЫЙ. Ведь Вы сами придумали такой синтаксис!

Зато Форт более чем искупает это упущение, позволяя Вам компилировать каждое определение по отдельности и в течение считанных секунд. Вы обнаруживаете свою ошибку достаточно быстро - когда Ваше определение не срабатывает. Кроме того, при желании Вы можете добавить в свои определения подходящие синтаксические проверки.

Кисть артиста не может защитить его от ошибки, художник сам будет судить об этом. Сковорода повара и рояль композитора остаются простыми и производительными. Зачем же позволять языку программирования пытаться быть умнее Вас?

Так является ли Форт высокоуровневым языком? По вопросу проверки синтаксиса он не проходит. По вопросу уровня абстрагирования и мощности он кажется языком `безграничного` уровня - поддерживающим все - от манипуляции с битами в порту вывода до задач бизнеса.

Решаете Вы. (Форту все равно.)

ЯЗЫК ПРОЕКТИРОВАНИЯ

Форт - язык для проектирования. Воспитаннику традиционной компьютерной науки такое утверждение кажется противоречивым. "Нельзя проектировать с помощью языка, с его помощью реализуют. Проектирование предваряет реализацию."

Опытные Форт-программисты с этим не соглашаются. Вы можете писать на Форте абстрактный, проектный код и все равно имеете возможность проверить его в любой момент, применяя преимущества разбиения на лексиконы. При продолжении разработки компонент может быть легко переписан на уровне ниже компонентов, которые его используют. Вначале слова могут печатать числа на Вашем терминале вместо управления шаговыми двигателями. Они могут печатать свои собственные имена только для того, чтобы сообщить Вам о своем выполнении. Они вообще могут ничего не делать.

Используя такую технологию, Вы можете писать простую, но проверяемую версию Вашей задачи, а затем успешно изменять и улучшать ее до тех пор, пока не достигнете своей цели.

Другим фактором, делающим возможным проектирование в коде, является то, что Форт, как и некоторые новейшие языки, сокращает последовательность разработки "редактирование- компиляция- тестирование- редактирование- компиляция- тестирование". Вследствие постоянной обратной связи среда окружения становится партнером в созидательном процессе.

Программист, использующий обычный язык, редко может достичь того продуктивного образа мышления, которое присуще артистам, если ничто не мешает течению их творческого процесса.

По этим причинам Форт-программисты тратят меньше времени на планирование, чем их коллеги классического толка - праведники планирования. Для них отсутствие такового кажется безрассудным и безответственным. Традиционные окружения вынуждают программистов планировать, поскольку традиционные языки не готовы к восприятию перемен.

К сожалению, человеческое предвидение ограничено даже при наилучших условиях. Слишком сильное планирование становится непродуктивным.

Конечно, Форт не исключает планирования. Он позволяет создавать прототипы. Конструирование прототипа - лучший способ планирования, так же, как и макетирование в электронике.

В следующей главе мы увидим, что экспериментирование проявляет себя более надежным в деле приближения к истине, нежели строительство догадок при планировании.

ПРОИЗВОДИТЕЛЬНЫЙ ЯЗЫК

Несмотря на то, что производительность не является главной темой этой книги, начинающий в Форте должен быть убежден в том, что преимущества языка не являются

чисто философскими. В целом Форт превышает все другие высокоуровневые языки по скорости работы, возможностям и компактности.

СКОРОСТЬ.

Несмотря на то, что Форт - интерпретирующий язык, он исполняет скомпилированный код. Поэтому он работает примерно в десять раз быстрее, чем интерпретирующий Бейсик.

Форт оптимизирован для исполнения слов с помощью техники, известной как "шитый код" [9],[10],[11]. Плата за разбиение на модули, состоящие из очень маленьких кусочков кода, сравнительно невелика.

Он не работает так же быстро, как ассемблерный код, поскольку внутренний интерпретатор (который обрабатывает список адресов, составляющих каждое определение через двоеточие) может отнимать до 50% времени исполнения слов-примитивов, в зависимости от типа процессора.

Но для больших задач Форт очень близко подходит к скорости ассмеблера. Вот три причины:

Первая и главная, Форт прост. Использование им стека данных значительно снижает затраты по производительности на передачу аргументов от слова к слову. В большинстве языков передача аргументов между модулями - одна из основных причин, по которым применение подпрограмм ограничивает их производительность.

Второе, Форт позволяет Вам определять слова либо на высоком уровне, либо на машинном языке. В любом случае нет необходимости в специальной вызывающей последовательности. Вы можете писать новое определение на высоком уровне и, убедившись в его правильности, переписать его на ассемблере без изменения какого-либо использующего его кода. В типичной задаче, быть может, 20% кода будет использоваться 80% времени.

Только наиболее часто используемые, критичные ко времени алгоритмы нуждаются в машинном кодировании. Форт-система сама во многом реализована на ассемблерных определениях, так что Вам нужно будет воспользоваться ассемблером лишь для нескольких специфических слов.

Третье, программы на Форте имеют тенденцию быть лучше спроектированными, чем те, что написаны целиком на ассемблере. Форт-программисты извлекают выгоду из способностей языка к созданию прототипов и испытывают несколько алгоритмов перед тем, как выбрать наиболее подходящий к их потребностям. Поскольку Форт поддерживает изменения, он может также быть назван языком оптимизации.

Форт не гарантирует быстроту исполнения. Он лишь дает программисту творческую среду, в которой можно разрабатывать быстродействующие программы.

ВОЗМОЖНОСТИ.

Форт может делать все, что могут другие языки - но обычно проще.

На нижнем уровне почти все Форт-системы имеют ассемблеры. Они поддерживают структурные операторы передачи управления для организации проверки условий и циклов, использующих технику структурированного программирования. Они обычно позволяют Вам писать подпрограммы обработки прерываний - Вы можете даже при желании писать их тело в высокоуровневом коде.



The best top-down designs of mice and young men.

Форт может быть написан для работы под управлением любой операционной системы, такой как RT-11, CP/M или MS-DOS - или, для тех, кто это предпочитает, Форт может быть написан как самодостаточная операционная система со своими драйверами терминала и дисков.

С помощью кросс-компилятора Форта или целевого компилятора вы можете создавать новые Форт-системы для того же или для разных компьютеров. Поскольку Форт написан на Форте, Вы имеете невиданную возможность переписывать операционную систему в зависимости от нужд Вашей задачи. Или Вы можете переместить различные версии задачи на ряд систем.

РАЗМЕРЫ.

Здесь имеются два соображения: размер корневой Форт-системы и размеры скомпилированных задач.

Ядро Форта является чрезвычайно гибким. Для встроенных применений часть Форта, необходимая для запуска программы, может уместиться всего в 1 КБайт. В полной инструментальной среде многозадачная Форт-система с интерпретатором, компилятором, ассемблером, редактором, операционной системой и другими утилитами поддержки занимает около 16 КБайт. При этом остается много места для задач. (А некоторые Форты на новых процессорах имеют 32-х разрядную адресацию, что позволяет писать невообразимо большие программы.)

Точно так же скомпилированные Форт-программы имеют очень маленький размер - обычно меньше эквивалентных программ на ассемблере. Причиной опять же является шитый код. Каждая ссылка на предварительно определенное слово, независимо от его мощности, использует всего два байта.

Одной из наиболее впечатляющих новых областей применения Форта является производство Форт-кристаллов, таких как Форт-микропроцессор Rockwell R65F11 [12]. На кристалле имеются не только аппаратные средства, но также исполняемая часть языка Форт и операционной системы для сложных применений.

Только архитектура Форта и его компактность делают возможным создание микропроцессоров, основанных на Форте.

ИТОГИ

Форт часто характеризуется как необычный, совершенно непохожий на любой другой популярный язык программирования - как по своей структуре, так и по философии. Однако Форт включает в себя многие из принципов, которыми щеголяют большинство современных языков. Структурированная разработка, модульность и упрятывание информации - среди ключевых слов сегодняшнего дня.

Некоторые новейшие языки близко подходят к духу Форта. Язык С, например, как и Форт, дает возможность программисту определять новые функции либо на С, либо на ассемблере. И, как и Форт, большая часть С определена в терминах функций.

Но Форт расширяет концепции модульности и упрятывания информации в большей степени, чем любой другой современный язык. Форт даже скрывает способ, которым вызываются слова и способ, по которому передаются локальные аргументы.

Результирующий код становится концентрированной смесью слов, чистейшим выражением абстрактного замысла. Как результат, Форт-программисты обычно продуктивнее, и пишут более плотные, эффективные и лучше управляемые программы.

Форт может не быть единственным возможным языком. Но я думаю, что подобный язык, если такая вещь возможна, был бы ближе к Форту, чем любой другой современный язык.

ЛИТЕРАТУРА

1. O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, `Structured Programming`, London, Academic Press, 1972.
2. Niklaus Wirth, "Program Development by Stepwise Refinement," `Communications of ACM`, 14, No.4(1971), 221-27.
3. W.P. Stevens, G.J. Myers, and L.L. Constantine, "Structured Design," `IBM Systems Journal`, Vol.13, No.2, 1974.
4. David L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules," `Communications of the ACM`, December 1972.
5. Barbara H. Liskov and Stephen N. Zilles, "Specification Techniques for Data Abstractions," `IEEE Transactions on Software Engineering`, March 1975.
6. David L. Parnas, "Designing Software for Ease of Extension and Construction," `IEEE Transactions on Software Engineering`, March 1979.
7. Dewey Val Shorre, "Adding Modules to FORTH," 1980 FORML Proceedings, p.71.
8. Mark Bernstein, "Programming in the Laboratory," unpublished paper, 1983.
9. James R. Bell, "Threaded Code", `Communications of ACM`, Vol.16, No.6, 370-72.
10. Robert B.K. DeWar, "Indirect Threaded Code," `Communications of ACM`, Vol.18, No.6, 331.
11. Peter M. Kogge, "An Architectual Trail to Threaded-Code Systems," `Computer`, March, 1982.
12. Randy Dumse, "The R65F11 FORTH Chip," `FORTH Dimensions`, Vol.5, No.2, p.25.

ГЛАВА 2

АНАЛИЗ

Любой, кто скажет Вам, что существует некоторое определенное количество фаз в цикле разработки программного обеспечения, будет глупцом.

И все же ...

ДЕВЯТЬ ФАЗ ЦИКЛА ПРОГРАММИРОВАНИЯ

Как мы видели, Форт интегрирует аспекты проектирования с вопросами реализации и поддержки. В результате этого упоминание о "типичном цикле разработки" звучит примерно так же, как и упоминание о "типичном шуме".

Однако любой подход лучше, чем отсутствие подхода и, разумеется, некоторые из подходов разработаны лучше других. Вот цикл разработки, представляющий некий "средний" из наиболее успешных способов, применяемых в программных проектах:

`Анализ`

1. Установка требований и ограничений
2. Построение концептуальной модели решения
3. Оценка цены/графика работ/производительности

`Конструирование`

4. Предварительная разработка
5. Детальная проработка
6. Реализация

`Использование`

7. Оптимизация
8. Устранение ошибок и отладка
9. Поддержка

В этой книге мы рассматриваем первые шесть стадий цикла, делая акцент на анализе, проектировании и реализации.

В Форт-проекте перечисленные фазы появляются на нескольких уровнях. Глядя на проект в самой широкой перспективе, можно сказать, что каждый из трех шагов может занимать месяц или более. Одна стадия сменяет другую, как времена года.

Но Форт-программисты применяют те же самые фазы при определении каждого слова. В этом случае цикл повторяется в течение минут.

Разработка программы с таким быстрым повторением программного цикла известна как использование "итеративного подхода".

ИТЕРАТИВНЫЙ ПОДХОД

Итеративный подход был красноречиво описан Кимом Харрисом [1]. Он начинает с определения научного метода:

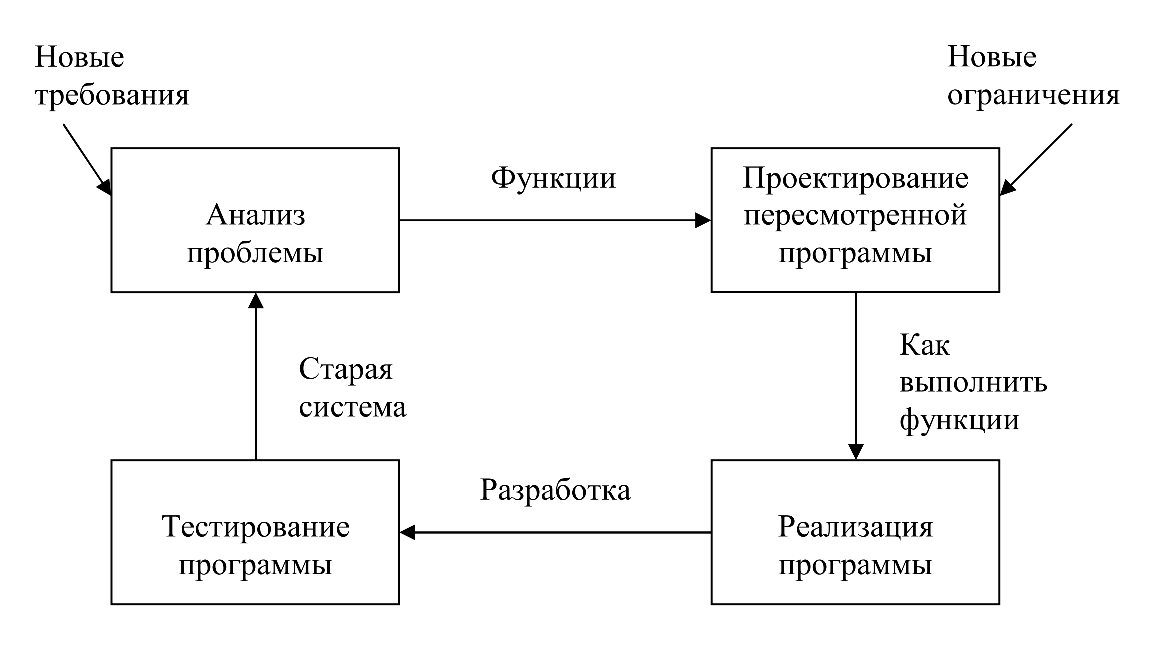
... бесконечный цикл открытий и улучшения. Он вначале изучает естественную систему и собирает сведения о ее поведении. Затем наблюдения моделируются для выработки теории об естественной системе. Далее инструменты анализа применяются к модели, что позволяет выдать предсказания о поведении реальной системы. Проводятся эксперименты с тем, чтобы сравнить истинное поведение с предсказанным.

Природная система вновь изучается, и модель пересматривается. `Целью` метода является выработка модели, которая в точности предсказывает все обозримое поведение естественной системы.

Затем Харрис применяет научный метод к циклу разработки программ, проиллюстрированному на рис. 2-1:

1. Проблема анализируется для выявления функций, требуемых для ее решения.
2. Принимаются решения о том, как достигнуть реализации этих функций с использованием доступных ресурсов.
3. Пишется программа, с помощью которой предпринимается попытка реализации проекта.
4. Программа проверяется на предмет правильности реализации функций.

Рис.2-1. Итеративный подход к циклу разработки программ, из статьи "Философия Форта" Кима Харриса. ("The FORTH Philosophy," by Kim Harris, Dr. Dobb's Journal.)



М-р Харрис добавляет:

Разработка программ на Форте в первую очередь направлена на поиск самого простого решения данной проблемы. Это достигается реализацией выбранных частей задачи раздельно и игнорированием возможно большего числа ограничений. Затем вводятся одно или несколько ограничений, и программа модифицируется.

Превосходным свидетельством в пользу модели проектирования вида разработка/проверка является эволюция. От протоплазмы к головастику и к человеку, каждый вид на этом пути состоит из функциональных, жизнеспособных существ. Создатель явно не выглядит проектировщиком "сверху-вниз".

СОВЕТ
Начинайте с простого. Пусть оно заработает. Поймите, что
именно Вы пытаетесь получить. Добавляйте сложность
понемногу, насколько это необходимо для выполнения

ОБЬЕМ ПЛАНИРОВАНИЯ

В девяти фазах, перечисленных в начале этой главы, мы выделили пять шагов, предпринимаемых `перед` "разработкой". Раньше в Главе 1 мы увидели, что чрезмерное увлечение планированием одновременно и сложно, и бесцельно.

Ясно, что нельзя предпринимать важный программный проект - независимо от языка - без некоторого количества планирования.
Какое же количество оптимально?

Многие Форт-программисты выражали глубокую признательность дотошному подходу к планированию Дейва Джонсона. Джонсон работает супервизором в фирме Moore Products Co. В Спрингсхаузе, штат Пенсильвания. Фирма специализируется на приложениях промышленной автоматизации и управлении процессами.
Дейв использует Форт с 1978 года. Вот как он описывает свой подход:

По сравнению со многими другими пользователями Форта я полагаю, что необходим более формальный подход. Я научился этому, пройдя тяжелый путь. Моя недисциплинированность в первые годы ныне как призрак посещает меня.

Мы используем два инструмента для начала проектирования новых продуктов: функциональное описание и проектное описание. Наш отдел продаж и применений выдает функциональное описание в соответствии с контрактом заказчика.

Когда мы договорились о том, что мы собираемся делать, функциональное описание приходит в наш отдел. С этого момента мы прорабатываем проект и выдаем проектное описание.

До этого момента наш подход не отличается от программирования на любом другом языке. Но с Фортом мы подходим к проектированию несколько по-другому. На Форте не надо проделывать 95% проектной работы до начала кодирования, достаточно 60% перед входом в итеративный процесс.

Типичный проект может состоять в функциональном усовершенствовании одного из наших продуктов. К примеру, у нас есть интеллектуальный терминал с дисководами и

нам нужно реализовать определенные протоколы для связи с другим устройством. Проектирование протоколов, связи с дисплеями, интерфейса с оператором и т.д. может занять несколько месяцев. Функциональное описание занимает месяц; проектное описание берет месяц; кодирование отнимает три месяца; сборка и проверка продлится еще месяц.

Это - типичный цикл. Один проект занимал почти два года, из которых шесть или семь месяцев - обоснованно. Когда мы пять лет назад принялись за Форт, это было не так. Как только я получал функциональное описание, я немедленно начинал кодирование. Я использовал нечто среднее между разработкой сверху-вниз и снизу-вверх, в основном определяя структуру и, по мере необходимости, некоторые низкоуровневые слова, затем вновь возвращаясь к структуре.

Причиной такого подхода было огромное давление в сторону демонстрации чего-нибудь начальству. Мы крутились, как заведенные, никогда не описывая того, что делали. Через три года нам понадобилось вернуться назад и пытаться модифицировать этот код, без какой-либо документации. Форт стал недостатком, поскольку позволил нам начать слишком рано. Было забавно зажигать огоньки и заставлять жужжать дисководы. Но мы не проходили через горнило проектной работы. Как я сказал, наши "свободные духи" вернулись к нам в виде призраков.

Ныне для новых программистов мы установили требование: полное проектное описание, которое определяет в деталях все высокоуровневые Форт-слова - задачи, которые Ваш проект должен выполнять. Нет больше чтения нескольких страниц функционального описания, ответа на них, чтения еще нескольких, ответа на них и т.д.

Ни один из живущих программистов не любит документирование. Имея предварительный проект, мы имеем возможность просмотреть все через несколько лет и вспомнить, что мы делали. Я должен заметить, что во время проектной фазы надо написать некоторое количество кода для проверки конкретных мыслей. Но этот код может и не быть частью законченного продукта. Идея состоит в том, чтобы обдумать Ваш проект.

Джонсон советует нам составлять полное проектное описание до начала кодирования, за исключением необходимых предварительных тестов. Следующее интервью поддерживает эту точку зрения и добавляет новые аргументы.

Джон Телеска стал независимым программистом-консультантом с 1976 года, специализируясь на традиционных для академического исследовательского оборудования задачах. Ему нравится снабжать исследовательские инструменты "как раз тем, что способна дать новейшая технология". Телеска работает в Рочестере, штат Нью-Йорк:

Я вижу две фазы в процессе разработки программ. Первая состоит в том, чтобы убедиться в моем понимании проблемы. Вторая заключается в реализации, включая отладку, верификацию и т.д.

Моя цель в фазе первой - операционное описание. Я начинаю с описания задачи, и по мере работы над ним получаю операционное описание - мое понимание того, как проблема преобразуется в решение. Чем лучше понимание, тем полнее решение. Я добиваюсь завершенности; чувства того, что больше нет вопросов, на которых нет ответа на бумаге.

Я понял, что чем больше времени в каждом проекте я трачу на первую фазу, тем больше беспокойство большинства моих клиентов. Ограничивающим фактором является

то количество необходимого для этого периода времени, на которое мне удастся убедить клиента. Потребители обычно не знают характеристик работы, которую они хотят получить. И у них нет денег - или у них нет чувства того, что они есть - для траты на хорошие описания. Частью моей работы является убедить их в том, что не иметь их в конце концов обойдется дороже и по деньгам, и по времени.

Часть первой фазы тратится на изучение возможных решений. При написании спецификаций всплывают неопределенности. Я пытаюсь быть настолько неопределенным по отношению к неопределенностям, насколько это возможно. Например, им хочется собирать 200000 отсчетов в секунду с определенной точностью. Я в первую очередь должен проверить, что это вообще возможно на имеющимся у них оборудовании. При этом мне нужно для проверки возможностей написать заплатку из кода.

Другим аргументом в пользу описаний является прикрытие самого себя. В случае, если задача работает по спецификации, но не полностью удовлетворяет потребителя, то за это несет ответственность он. Если ему захочется большего, то мы должны договориться заново. Но я думаю, разработчик отвечает за то, чтобы сделать все возможное для выдачи операционного описания, которое будет работать к удовлетворению покупателя.

Я думаю, что есть консультанты, уступающие давлению клиента и ограничивающие время, затрачиваемое на описания из страха потерять работу. Но в таких ситуациях ни для кого не бывает счастливого конца.

Мы вернемся к интервью с Телеской через мгновение.

ОГРАНИЧЕНИЯ ПЛАНИРОВАНИЯ

Опыт научил нас намечать свой путь перед началом кодирования. Но у планирования есть и определенные ограничения. Следующие интервью дают различные оценки для количества планирования.

Несмотря на предпочтение Телеской хорошо спланированного проектирования, он рекомендует выбирать между подходом сверху-вниз и снизу-вверх в зависимости от ситуации:

В двух недавних проектах, содержавших много технической стыковочной работы, я проделал все снизу-вверх. Я перемолол кипу бумаг с данными и техническими описаниями мелких щелей операционной системы, с которой я имел дело.

Большую часть времени я чувствовал себя потерянным, не понимая, зачем я вообще взялся за эту работу. Затем я достиг некоторой критической массы известного вида и начал составлять вместе маленькие программы, что приводило к возникновению маленьких вещей. Я продолжал, снизу-вверх, пока не достиг уровня целевого приложения.

Мой дух проектировщика "сверху-вниз" был напуган этой процедурой. Но я столько раз наблюдал себя успешно проходящим через этот процесс, что не могу сбросить его со счетов по какой-нибудь педагогической причине. И всегда бывает эта трудная стадия, которую, кажется, не преодолеет никакое количество линейного мышления.

Программирование кажется гораздо более интуитивным, чем мы, им занимающиеся, склонны говорить друг другу. Я думаю, что когда задача порождает такое чувство потерянности, мне надо работать снизу-вверх. Если я ощущаю себя на своей территории, то тогда, по-видимому, изберу более традиционный, книжный подход.

А вот еще одна точка зрения:

Во время нашего интервью Майкл Старлинг из фирмы Union Carbide делал последние штрихи в двух программах, касающихся конфигурируемых пользователем лабораторных и производственных систем автоматизации. Для опытной фабричной системы Старлинг разработал как аппаратуру, так и программное обеспечение в соответствии с известными требованиями; на лабораторной системе он также сам определял требования.

Его усилия увенчались полным успехом. Для одного проекта новая система стоит лишь 20% от цены аналогичной системы и требует дней, а не месяцев, для установки и конфигурирования.

Я спросил его, какую технику он применял для поддержки проекта.

Для обеих задач требовалось большое количество проектирования. Тем не менее я не следовал традиционным методам анализа. Я осуществил следующие шаги:

Во-первых, я четко отследил ограничения задачи.

Во-вторых, я определил, какими должны быть минимальные функциональные куски - программные подсистемы.

В-третьих, я сделал каждый кусочек, составил их вместе, и система стала работать. Затем я спросил пользователей: "Это удовлетворяет вашим требованиям?" В чем-то оно не удовлетворяло, причем таким образом, каким ни пользователи, ни разработчики проекта не могли предугадать.

К примеру, разработчики не осознавали, что начальное описание не предусматривало приятные, ориентированные на человеческое восприятие графические картинки. При работе с интерактивной графикой первой версии пользователи применяли фиксированные масштабы графики и получали странные картинки.

Поэтому, даже уже после того, как был разработан базовый алгоритм графики, мы осознали, что необходимо было делать автоматическое масштабирование. Мы вернулись назад, изучили, как человеческие существа чертят графики, и написали функцию нижнего уровня, которая подсчитывают данные по осям X и Y и как они разместятся на графике.

После этого мы осознали, что не все взятые данные будут представлять интерес для экспериментаторов. Поэтому мы добавили возможность увеличения отдельных кусков.

Такой итеративный подход позволил получить более четкий и лучше продуманный код. Мы выявили основную линию целей и построили минимальную систему, отвечающую известным требованиям пользователя. Затем мы покопали наш программистский опыт для его улучшения и установили, что из нужного заказчики позабыли при составлении спецификации. Пользователи, в основном, новых идей не придумали. Это сделали программисты, и они отделяют эти идеи от идей пользователей.

Постановка задачи оказалась улицей с двусторонним движением. В некоторых случаях они получали такие вещи, о которых и не думали, что они возможны на таком маленьком компьютере - например, применение цифровых фильтров и процессоров сигналов к данным.

Одним из свойств Форта, сделавших возможным такой подход, является то, что примитивы легко тестируются. Требуется некоторый опыт работы с Фортом, чтобы получать от этого преимущества. Ребята с традиционным воспитанием хотят написать за своим столом десять страниц кода, потом сесть и ввести их, и ожидают, что это будет работать.

Вот вкратце мой подход: я пытаюсь установить, что нужно пользователям, но в то же время осознаю неполность этих сведений. Затем я держу их вовлеченными в проект во время реализации, поскольку они должны выполнять роль экспертов. Когда они видят результат, это им приятно, поскольку известно, что их идеи использованы.

Итеративный подход позволяет достичь наивысших результатов при создании хорошего решения для реальной проблемы. Он может не всегда дать Вам объявленную заранее стоимость программного обеспечения. Путь решения может зависить от Ваших приоритетов. Запомните:

Хорошее
Быстрое
Дешевое

Выбирайте любые два качества!

Как говорит Старлинг, Вы как не знаете толком, что делаете, до тех пор, пока один раз это не пройдете. Мой опыт показывает, что наилучший способ написать программу - это написать ее дважды. Выкиньте первую версию, приняв ее за набросок.

Питер Кожж входит в основной технический состав подразделения федеральных систем фирмы IBM, Освего, штат Нью-Йорк:

Одним из ключевых преимуществ, которые я нахожу в Форте, является то, что он позволяет мне быстро создавать прототипы задачи без колокольного звона и свистопляски, зачастую с существенными ограничениями, однако в виде, достаточном для запуска "человеческого интерфейса" из подручных средств.

Когда я строю прототип, я делаю это с твердой уверенностью в том, что не использую ни строчки из текста прототипа в конечной программе. Эта вынужденная "переделка" почти всегда приводит к значительно более простым и более элегантным законченным программам, даже если последние написаны на чем-то, отличном от Форта.

Каковы наши выводы? В окружении Форта планирование необходимо. Но оно должно быть коротким. Тестирование и построение прототипов - наилучшие пути для понимания того, что именно действительно нужно.

Одно слово в предостережение руководителям проектов: если Вы наблюдаете за любым опытным Форт-программистом, то не надо беспокоиться насчет того, что он тратит слишком много времени на планирование. Так что следующий совет имеет две версии:

СОВЕТ

Для новичков в Форте (с "традиционным" воспитанием):
Сокращайте до минимума фазу анализа.
Для приверженцев Форта (без "традиционной" подготовки):
Воздерживайтесь от кодирования столь долго, сколько сможете выдержать.

Или, как мы упоминали в первой главе:

СОВЕТ

Планируйте изменения (проектируя компоненты, которые могут быть изменены).

Или просто:

СОВЕТ

Делайте прототипы.

ФАЗА АНАЛИЗА

В оставшейся части этой главы мы обсудим фазу анализа. Анализ - это организованный путь к пониманию и документированию того, что должна делать программа.

Для простой программы, которую Вы пишете для себя меньше чем за час, фаза анализа может занять около 250 микросекунд. В другой крайности некоторые проекты займут много человеко-годов. В таком проекте фаза анализа определяет успех всего проекта.

Мы указывали на три части фазы анализа:

1. Установка требований и ограничений
2. Построение концептуальной модели решения
3. Оценка цены/графика работ/производительности

Давайте вкратце опишем каждую часть:

УСТАНОВКА ТРЕБОВАНИЙ.

Первым шагом является выяснение того, что должна делать задача. Покупатель или вообще тот, кто хочет иметь систему, должен предоставить "описание требований". Это – честный документ, перечисляющий минимальные возможности конечного продукта.

Аналитик может также делать дальнейшие пробы, проводя беседы и рассылая вопросники пользователям.

УСТАНОВКА ОГРАНИЧЕНИЙ.

Следующий шаг заключается в определении ограничивающих факторов. Насколько важна скорость? Сколько доступно памяти? Как быстро нужно получить разработку?

Независимо от утонченности нашей технологии, программисты всегда будут биться об ограничения. Возможности системы необъяснимым образом уменьшаются со временем. Дисководы с двойной плотностью записи, однажды послужившие ответом на мои молитвы, ныне не удовлетворяют меня. Двусторонние, с двойной плотностью дисководы, которые я заведу следующими, покажутся безграничными - на время. Я слышал жалобы на тесноту от людей с 10-ю мегабайтовыми жесткими дисками.

Где бы ни ощущалась нехватка чего-либо - а она всегда будет - следует делать компромиссы. Лучше использовать фазу анализа для противостояния большинству ограничений и принятия решений о том, какие нужны компромиссы.

С другой стороны, во время анализа Вы `не` должны принимать во внимание другие типы ограничений, преодолевая их постепенно во время реализации по типу того, как растирают комки в тесте.

Во время анализа следует принимать во внимание те типы ограничений, которые могут повлиять на подход в целом. Отложить следует те, которые могут быть учтены во время итеративного улучшения спланированного программного проекта.

Как мы слышали в предыдущих интервью, выявление `аппаратных` ограничений часто требует написания некоторого количества пробного кода и испытаний. Выявление ограничений со стороны `покупателя` обычно сводится к заданию ему вопросов или получению письменных обзоров. "Насколько быстро Вам необходимо то-то и то-то, в размере от одного до десяти?" и т.д.

ПОСТРОЕНИЕ КОНЦЕПТУАЛЬНОЙ МОДЕЛИ РЕШЕНИЯ.

Концептуальная модель - это воображаемое решение проблемы. Это - взгляд на то, как система `должна` работать. Это – ответ на все требования и ограничения.

Если определение требований звучит как "нечто, на чем можно стоять при покраске потолка", то описанием концептуальной модели будет "устройство, стоящее свободно (так, что можно красить в середине комнаты), с несколькими ступенями, отделенными друг от друга одинаковыми интервалами (так, что можно забираться вверх и вниз) и имеющее маленькую полку на вершине (для банки с краской)".

Концептуальная модель, однако - это не совсем проект. Проект начинается с описания того, как система `в действительности` работает. В проекте должен был бы начать появляться образ лестницы со ступеньками.

Форт несколько размывает это различие, поскольку все определения пишутся в концептуальных терминах, используя лексиконы компонентов нижних уровней. На самом деле, позже в этой главе мы будем использовать `псевдокод` на Форте для описания реализаций концептуальной модели.

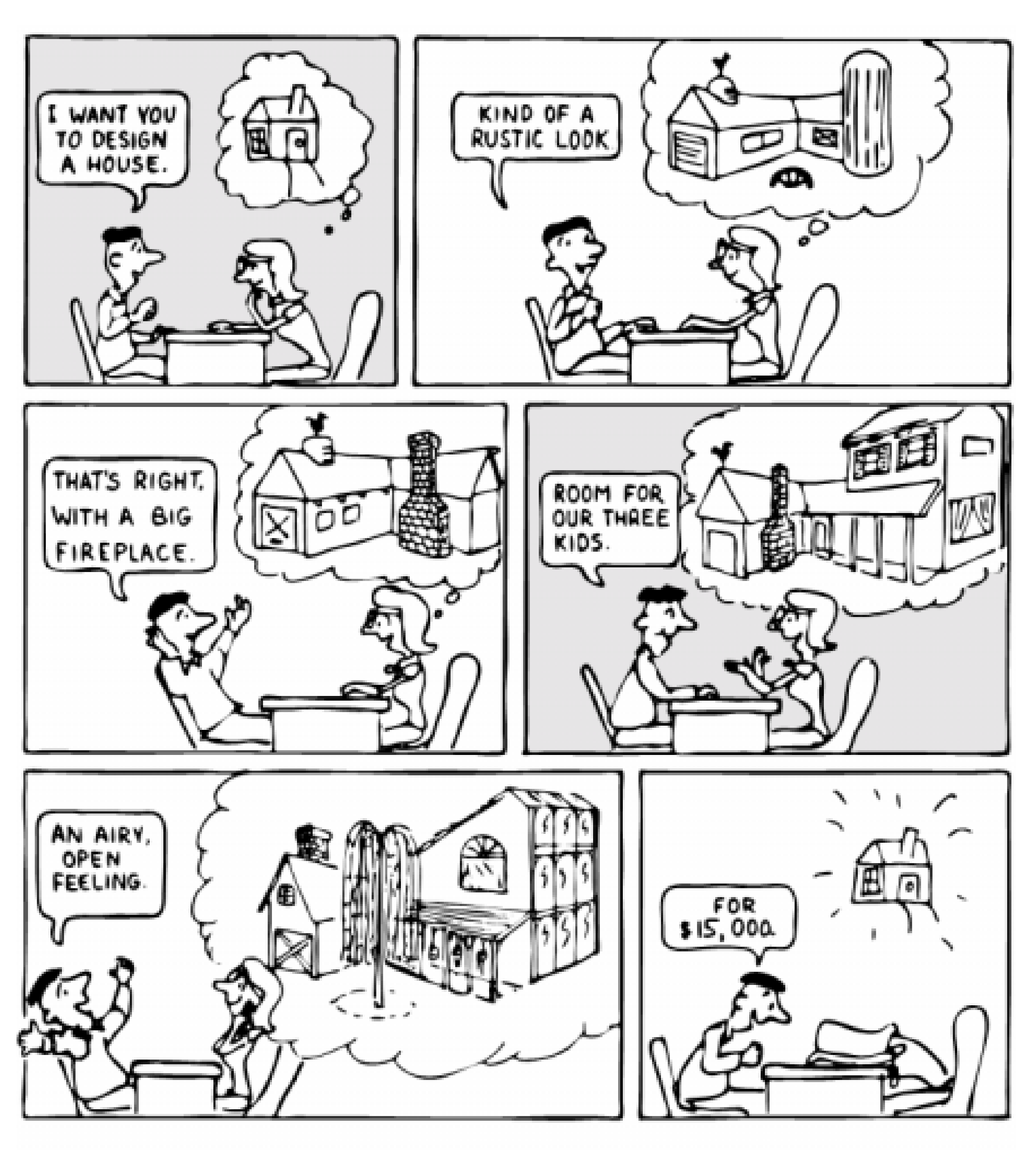
Несмотря на это, полезно делать различие. Концептуальная модель более гибка, чем проект. Легче войти в рамки требований и ограничений в модели, чем в проекте.

СОВЕТ
Старайтесь построить солидную концептуальную модель перед
началом проектирования.

Анализ состоит из расширения описания требований до уровня концептуальной модели. Технология включает двухстороннюю связь с покупателем в попытках достичь успеха в описании модели.

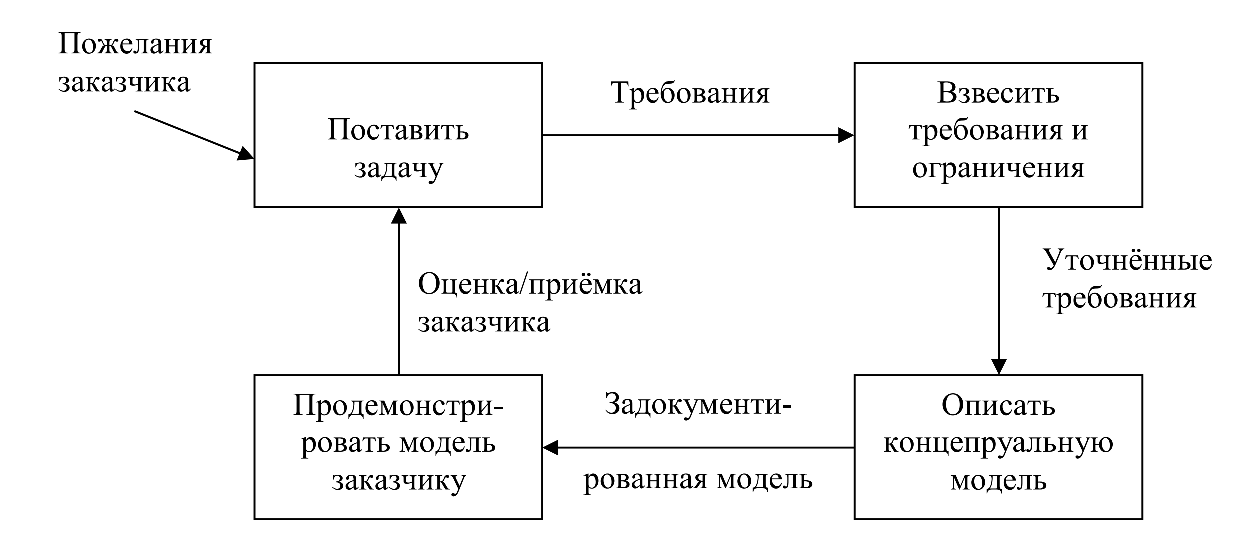
Как и весь цикл разработки, в фазе анализа лучше всего итеративный подход. Каждое новое требование будет пытаться добавить что-то в Вашу мысленную модель. Ваша работа состоит в жонглировании всеми этими требованиями и ограничениями до тех пор, пока Вы не сплетете основу, которая им удовлетворяет.

Рисунок 2-2 иллюстрирует итеративный подход к фазе анализа. Последний шаг является одним из наиболее важных: показать задокументированную модель покупателю. Используйте любые необходимые средства связи - диаграммы, таблицы или картинки - для доведения Вашего понимания до заказчика и получайте необходимую обратную связь. Даже если Вы проделаете весь цикл сто раз, эти усилия не потеряют свою ценность.



Refining the conceptual model to meet requirements and constraints.

Рис.2-2. Итеративный подход к анализу.



В следующих двух разделах мы исследуем три вида техники для определения и описания концептуальной модели:

1. определение интерфейсов
2. определение правил
3. определение структур данных.

ОПРЕДЕЛЕНИЕ ИНТЕРФЕЙСОВ

СОВЕТ
Первое и самое важное: концептуальная модель должна описывать интерфейсы системы.

Телеска:

"Описание" обычно имеет дело с понятием ЧТО. В самом лучшем случае оно должно показывать то, на что система будет похожа для пользователя - можно назвать это руководством пользователя. Мне кажется, что я пишу больше замечаний по взаимодействию с человеком - как это выглядит снаружи - чем по той части, которая в действительности выполняет работу. К примеру, я составляю полное описание обработки ошибок для того, чтобы показать, что происходит при определенных событиях. К несчастью, эта часть заодно отнимает и большую часть времени при реализации.

В настоящее время я работаю над твердотельным таймером для промышленной стиральной машины. В данном случае интерфейс с пользователем не так сложен. Что сложно, так это интерфейс со стиральной машиной, в котором я должен придерживаться требований заказчика и той документации, которую он может предоставить.

Интерфейс - это руки и ноги для продукта. На ранней стадии я не делаю различия между аппаратурой и программным обеспечением. Они оба могут взаимозаменяться при реализации.

Процесс проектирования аппаратуры и процесс проектирования программ аналогичны. Путь, по которому я разрабатываю аппаратуру - это представление ее в образе черного ящика.

Передняя панель - это вход и выход. То же самое можно сделать и с программным обеспечением. Я использую всякие приемы, диаграммы и т.п. для того, чтобы показать покупателю, как выглядят входы/выходы, использую описания того, что продукт должен делать. Но, параллельно, в воображении, я представляю как он должен быть реализован. Вычисляю, насколько эффективно я могу это сделать. Так что для меня это не черный ящик, это серый ящик.

Проектировщик должен быть способен видеть внутри черных ящиков. При проектировании системы получаются различные модули, я стараюсь сделать их взаимосвязь настолько рациональной и настолько маленькой, насколько возможно. Но всегда где-то находишь, а где-то теряешь, поскольку изменяешь идеалу с компромиссами.

Для самого документа я использую ДПД [диаграммы потоков данных, которые мы обсудим позже] и любые другие виды представления, которые я могу продемонстрировать клиентам.

Я показываю им сколько возможно много диаграмм для пояснения своего понимания. Я обычно не использую их, когда дело доходит до реализации. Текст должен быть полноценен даже и без ссылок на диаграммы.

СОВЕТ

Решайте вопросы обработки ошибок и исключительных случаев заранее, при определении интерфейсов.

Верно то, что при написании программ для себя программист зачастую может сконцентрироваться на правильности работы кода при `нормальных` условиях и побеспокоиться об обработке ошибок позже. Однако при работе на других обработка ошибок должна быть проработана в первую очередь. Эта та область, которая часто выпадает из поля зрения начинающих программистов.

Причина важности принятия решений по обработке ошибок на этой стадии состоит в широком спектре методов их обработки. Ошибка может быть:

* игнорируемая
* устанавливающая флаг индикации ошибки при продолжении процесса
* немедленно останавливающая процесс
* вызывающая процедуру для корректировки задачи и продолжения работы.

Простор для возникновения серьезной коммуникационной бреши открывается в том случае, если уровень сложности при обработке ошибок заранее не отслежен. Очевидно, что этот выбор чрезвычайно сильно отражается на проекте и реализации задачи.

СОВЕТ

Разрабатывайте концептуальную модель, представляя себе, как данные проходят, и какие действия над ними производятся в частях модели.

Дисциплина, называемая `структурный анализ` [2], предлагает некоторые способы описания интерфейсов таким образом, что их легко поймут Ваши клиенты. Один из таких способов, упоминавшихся Телеской, называется "диаграммой потоков двнных" (ДПД).

Диаграмма потоков данных, типа изображенной на рис. 2-3, подчеркивает преобразования данных при их прохождении через систему. Круги представляют собой "преобразования", функции, воздействующие на информацию. Стрелки показывают входы и выходы преобразований.

Рис.2-3. Диаграмма потоков данных.

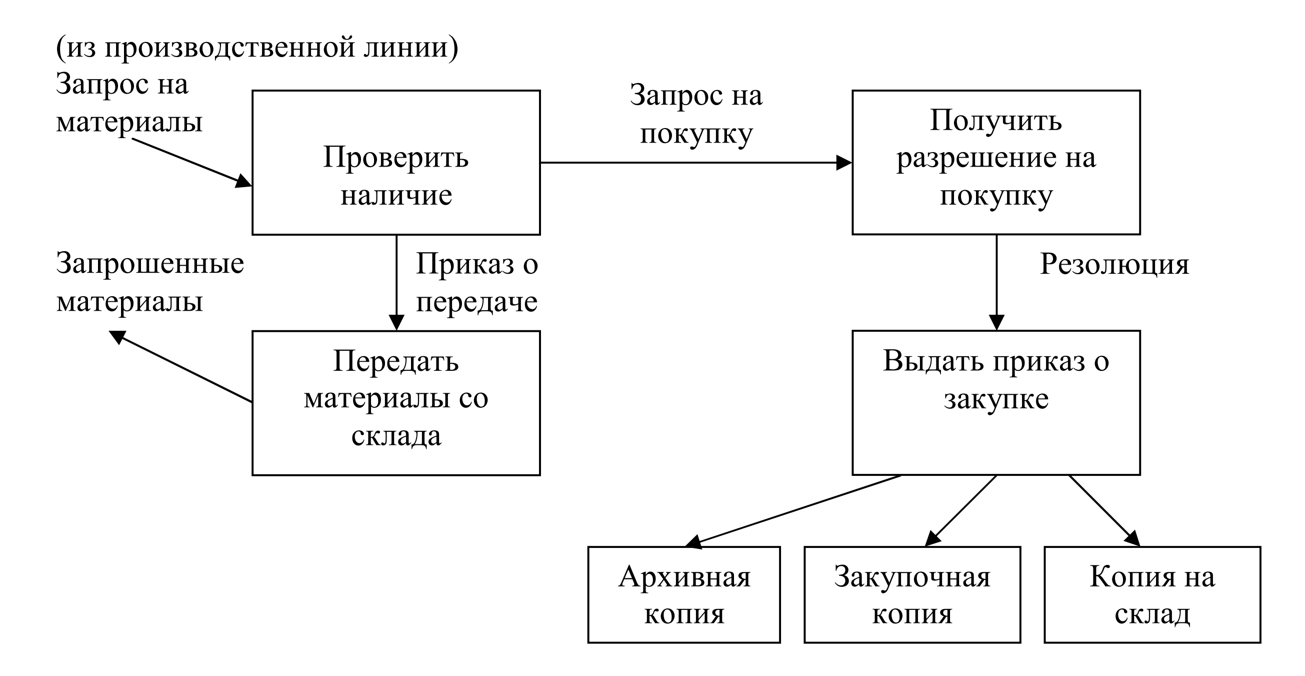


Диаграмма живописует застывший момент в работе системы. Она оставляет без внимания инициализацию, структуры циклов и другие детали программирования, которые зависят от времени.

При использовании ДПД достигаются три плюса:

Во-первых, они дают сведения заказчику при помощи простых, прямых терминов. Если он соглашается с содержимым Вашей диаграммы потоков данных, то Вы знаете, что понимаете проблему.

Во-вторых, они помогают Вам думать в терминах логических "ЧТО", без углубления в процедурные "КАК", что согласуется с философией упрятывания информации, обсужденной нами в предыдущей главе.

В-третьих, они фокусируют Ваше внимание на интерфейсах к системе и между модулями.

Несмотря на это, Форт-программисты редко используют ДПД иначе как для заказчика. Форт способствует Вашему мышлению в терминах концептуальной модели, а скрытое использование стека данных в Форте делает передачу данных между модулями настолько простой, что ее обычно можно принимать как само собой разумеющуюся. Это

происходит потому, что Форт, будучи использован правильно, приближается к уровню функционального языка.

Даже для тех, кто всего несколько дней знаком с Фортом, простые определения содержат по крайней мере столько же смысла, сколько диаграммы:

```
: ЗАПРОС  ( количество наименование -- )
      ПОД-РУКОЙ?  IF  ВЫДАТЬ  ELSE  ПЕРЕАДРЕСОВАТЬ  THEN ;
: ПЕРЕАДРЕСОВАТЬ    ЗАВЕРЕНО?  IF  ПОКУПКА  THEN ;
: ПОКУПКА    АРХИВНАЯ КОПИЯ   СКЛАДСКАЯ КОПИЯ
      ЗАКУПОЧНАЯ КОПИЯ ;
```

Это - псевдокод на Форте. Не было сделано никаких попыток конкретизировать, какие величины в действительности передаются через стек, поскольку это - детали реализации. Стековый комментарий для слова ЗАПРОС использован только для указания двух видов данных, необходимых для начала процесса.

(Если бы я проектировал эту задачу, я бы посоветовал, чтобы пользовательский интерфейс был представлен словом НУЖНО со следующим синтаксисом:

```
НУЖНО 50 ПОДШИПНИКИ
```

НУЖНО преобразует число в величину на стеке, переводит строку ПОДШИПНИКИ в номер запасной части, также оставляя его на стеке, затем вызывает ЗАПРОС. Такая команда могла бы быть определена только на самом удаленном от нижнего уровне.)

Джонсон из фирмы Moore Products Co. сказал несколько слов насчет Форт-псевдокода:

IBM использует строго документированный PDL (ЯПП – язык проектирования программ). Мы также используем PDL, хотя и называем его FDL, т.е. язык проектирования на Форте.

Наверное, расточительно иметь все эти стандарты, однако, если Вы знакомы с Фортом, то Форт сам может быть языком проектирования. Надо только отбросить так называемые "шумовые" слова: C@, DUP, OVER и т.п. и показывать только основной поток. Большинство работающих на Форте делают это интуитивно. Мы же делаем это целенаправленно.

Во время одного из наших интервью я спросил Мура, использовал ли он какой-нибудь вид диаграмм для планирования концептуальной модели или кодировал прямо на Форте. Вот его ответ:

Концептуальной моделью `является` Форт. Я годами учился думать в этом духе. Может ли кто-нибудь научиться так думать?

У меня есть нечестное преимущество. Я закодировал свой стиль программирования, и другие люди восприняли его. Я был поражен тем, что это случилось. И я чувствую, что это приятное преимущество, поскольку ведь это мой стиль другие пытаются скопировать. Могут ли они научиться думать так же, как я? Мне кажется, что могут. Это - всего лишь вопрос практики, а у меня практики больше.

ОПРЕДЕЛЕНИЕ ПРАВИЛ

Большая часть наших усилий при описании задачи концентрируется вокруг описания интерфейса. Но некоторые задачи требуют, чтобы Вы также определили набор правил для их решения.

Любое программирование опирается на правила. Обычно они настолько просты, что почти не играет роли, как Вы их выражаете: "если кто-либо нажмет кнопку, то зазвонит колокольчик".

Однако некоторые задачи содержат правила настолько сложные, что их нельзя выразить в нескольких фразах. Несколько формальных приемов могут быть Вам полезны для понимания и документирования таких сложных правил.

Вот пример. Имеющиеся требования описывают систему для вычисления платы за дальние телефонные переговоры. Вот объяснения заказчика о структуре платежей. (Я сам это придумал; не имею никакого представления о том, как в действительности телефонная компания вычисляет эти тарифы, знаю только, что всегда приходится переплачивать.)

Все тарифы исчисляются по-минутно, в соответствии с расстоянием в милях, плюс постоянная плата. Постоянная плата для прямого вызова по рабочим дням между 8 утра и 5 вечера составляет .30 (0.3 доллара) за 1-ю минуту и .20 за каждую последующую; кроме того, каждая минута увеличивается на .12 за 100 миль.

Постоянная плата в будни между 5 вечера и 11 вечера составляет .22 за первую минуту и .15 за каждую дополнительную; плата за расстояние - .10 на минуту за 100 миль. Постоянная плата для прямых вызовов поздно в будни от 11 вечера или в любое время в субботу, воскресенье или праздники составляет .12 за первую минуту и .09 для каждой следующей; наценка за расстояние на минуту .06 за 100 миль.

Если вызов требует помощи оператора, постоянная плата увеличивается на .90 назависимо от времени.

Это описание составлено на старом добром русском языке, и оно крайне многословно. Его трудно отслеживать и, смахивая на чердак, заваленный старым хламом, оно может даже содержать несколько ошибок.

При возведении концептуальной модели такой системы мы должны описать структуру платежей однозначным, удобным образом.

Первым шагом на пути к расчистке завала является вычленение независимых кусков информации - следуя применению правила ограниченной связности. Мы может сильно улучшить этот текст, разбив его на утверждения. Во-первых, это правило времени дня:

Вызовы в будни между 8 утра и 5 вечера идут по "полной" оплате. Вызовы в будни между 5 вечера и 11 вечера идут по "среднему" тарифу. Вызовы в будни от 11 вечера или по субботам, воскресеньям и праздникам оплачиваются по "низшему" тарифу.

Затем следует сама структура платежей, которая должна быть описана в терминах "тарифа за первую минуту", "тарифа за дополнительную минуту", "тарифа за расстояние" и "тарифа за работу оператора".

СОВЕТ
Разделите фрукты. (Не путайте яблоки с апельсинами.)

Однако эти эпистолярные утверждения все еще трудно читать.
Системные аналитики используют несколько способов для упрощения таких выражений: структурированный английский (русский) язык, деревья решений и таблицы решений. Давайте изучим каждую из этих технологий и прикинем их полезность в среде Форта.

СТРУКТУРИРОВАННЫЙ АНГЛИЙСКИЙ.

Структурированный английский (у нас - русский) - это вид структурированного псевдокода, при котором наше выражение для платежей будет читаться как-то вроде:

```
IF  полный тариф
    IF  прямой вызов
        IF  первая минута
            .30 + .12/100миль
        ELSE ( дополн. минута)
            .20 + .12/100миль
        ENDIF
    ELSE ( оператор)
        IF  первая минута
            1.20 + .12/100миль
        ELSE ( дополн. минута)
            .20 + .12/100миль
        ENDIF
    ENDIF
ELSE  ( не полный тариф)
    IF  средний тариф
        IF  прямой вызов
            IF  первая минута
                .22 + .10/100миль
            ELSE ( дополн. минута)
                .15 + .10/100миль
            ENDIF
        ELSE ( оператор)
            IF  первая минута
                1.12 + .10/100миль
            ELSE ( дополн. минута)
                .15 + .10/100миль
            ENDIF
        ENDIF
    ELSE  ( низкий тариф)
        IF  прямой вызов
            IF  первая минута
                .12 + .06/100миль
            ELSE ( дополн. минута)
                .09 + .06/100миль
            ENDIF
```

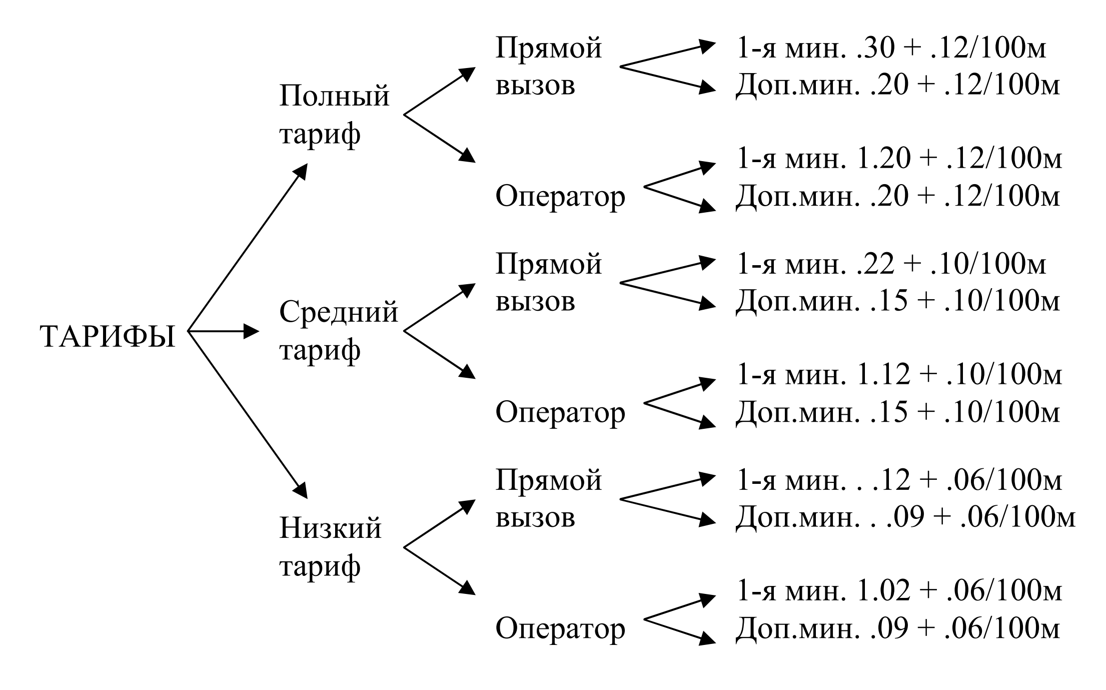
```
            ELSE  ( оператор)
                IF  первая минута
                    1.02 + .06/100миль
                ELSE ( дополн. минута)
                    .09 + .06/100миль
                ENDIF
            ENDIF
        ENDIF
    ENDIF
```

Это совершенно ужасно. Это трудно читать, еще труднее понимать и труднее всего писать. И, мало того, совершенно бесполезно для реализации. Я более не хочу об этом даже говорить.

ДЕРЕВО РЕШЕНИЙ.

Рисунок 2-4 представляет телефонные тарифы при помощи дерева решений. Дерево решений - простейший из всех метод "прослеживания" результата при некоторых условиях. По этой причине он может оказаться наилучшим представлением для демонстрации заказчику.

Рис.2-4. Пример дерева решений.



К сожалению, по дереву решений трудно "двигаться назад" для определения того, какие определенные условия приводят к каким результатам. Это затрудняет поиск путей для упрощения задачи. Дерево закрывает тот факт, что дополнительные минуты стоят одинаково, независимо от помощи оператора. Этот факт не виден на дереве.

ТАБЛИЦА РЕШЕНИЙ.

Таблица решений, описанная далее, дает наиболее приемлемое графическое представление сложных правил для программиста, и, возможно, также и для заказчика. Рисунок 2-5 показывает нашу структуру платежей в форме таблицы решений.

На рис.2-5 присутствуют три измерения: тарифная скидка, присутствие оператора и начальная/дополнительная минута.

Рис.2-5. Таблица решений.

	Полный тариф		Средний тариф		Низкий тариф	
	1-я минута	Доп. мин.	1-я минута	Доп. мин.	1-я минута	Доп. мин.
Прямой вызов	0,30+ 0,12/100км	0,20+ 0,12/100км	0,22+ 0,10/100км	0,15+ 0,10/100км	0,12+ 0,06/100км	0,09+ 0,06/100км
Оператор	1,20+ 0,12/100км	0,20+ 0,12/100км	1,12+ 0,10/100км	0,15+ 0,10/100км	1,02+ 0,06/100км	0,09+ 0,06/100км

Описание задачи более чем в двух измерениях может быть немного замысловатым. Как можно видеть, эти дополнительные измерения могут быть нарисованы на бумаге как подизмерения внутри наружных. Все условия этих подизмерений появляются внутри каждого из условий наружных измерений. Как мы увидим, в программе может быть легко реализовано любое количество измерений.

Все описанные нами приемы заставляют Вас анализировать, какие условия соответствуют каким измерениям. Во время такого разбиения применяются два правила:

Первое, все элементы каждого измерения должны быть взаимно исключающими. Вы не должны ставить "1-ю минуту" в то же измерение, что и "прямой вызов", поскольку они не взаимоисключают друг друга.

Второе, в каждом измерении должны быть перечислены все возможности. Если бы существовал иной тариф для звонков с 2 до 2.05 ночи, то таблицу пришлось бы увеличить.

Но наши таблицы решений имеют свои индивидуальные преимущества. Таблица решений не только хорошо читается клиентом, но также помогает разработчику несколькими способами:

`Преобразуемость в реальный код`. Это особенно верно для Форта, в котором таблицы решений легко реализуются в форме, очень близкой к рисунку.

`Способность к отслеживанию логики в обратную сторону`. Найдите условие и смотрите, какие факторы его создали.

`Более ясное графическое представление`. Таблицы решений служат лучшим инструментом для понимания как для аналитика, так и для разработчика.

В отличие от деревьев решений, таблицы решений группируют вместе `результаты` с помощью осмысленной графики. Визуализация идей помогает в понимании проблем, особенно тех, которые слишком сложны для выражения линейным образом.

К примеру, на рисунке 2-5 ясно видно, что плата за дополнительные минуты не зависит от вмешательства оператора. С осознанием этого мы можем нарисовать упрощенную таблицу, как показано на рис. 2-6.

Рис.2-6. Упрощенная таблица решений.

<table>
<tr><td></td><td colspan="2">Полный тариф</td><td colspan="2">Средний тариф</td><td colspan="2">Низкий тариф</td></tr>
<tr><td></td><td>1-я минута</td><td>Доп. мин.</td><td>1-я минута</td><td>Доп. мин.</td><td>1-я минута</td><td>Доп. мин.</td></tr>
<tr><td>Прямой вызов</td><td>0,30+
0,12/100км</td><td rowspan="2">0,20+
0,12/100км</td><td>0,22+
0,10/100км</td><td rowspan="2">0,15+
0,10/100км</td><td>0,12+
0,06/100км</td><td rowspan="2">0,09+
0,06/100км</td></tr>
<tr><td>Оператор</td><td>1,20+
0,12/100км</td><td>1,12+
0,10/100км</td><td>1,02+
0,06/100км</td></tr>
</table>

Легко так увлечься каким-нибудь аналитическим инструментом, что позабыть о самой задаче. Аналитик должен не только извлечь все возможности проблемы до N-го измерения, как рекомендуют некоторые из виденных мною авторов по структурному анализу.

Такой подход только увеличивает количество участвующих деталей. Решающий задачу должен также пытаться упростить ее.

СОВЕТ

Вы не понимаете проблему до тех пор, пока не можете упростить ее.

Если целью анализа является не только понимание, но и упрощение, тогда нам, быть может, следует проделать дополнительную работу.

Наша пересмотренная таблица решений (рис. 2-6) показывает, что плата за милю зависит только от того, является ли тариф полным, средним или низким. Другими словами, в таблице играет роль только одно из трех измерений. Что случится, если мы разобьем таблицу на две, как на рис.2-7?

Рис.2-7. Разбиение таблицы решений.

<table>
<tr><td></td><td></td><td colspan="2">Полный тариф</td><td colspan="2">Средний тариф</td><td colspan="2">Низкий тариф</td></tr>
<tr><td></td><td></td><td>1 мин.</td><td>Доп. мин.</td><td>1 мин.</td><td>Доп. мин.</td><td>1 мин.</td><td>Доп. мин.</td></tr>
<tr><td rowspan="2">Плата за соедине-ние</td><td>Прямой вызов</td><td>0,30</td><td rowspan="2">0,20</td><td>0,22</td><td rowspan="2">0,15</td><td>0,12</td><td rowspan="2">0,09</td></tr>
<tr><td>Оператор</td><td>1,20</td><td>1,12</td><td>1,02</td></tr>
<tr><td colspan="2">ПЛЮС</td><td colspan="2"></td><td colspan="2"></td><td colspan="2"></td></tr>
<tr><td colspan="2">Плата за расстояние</td><td colspan="2">0,12/100км</td><td colspan="2">0,10/100км</td><td colspan="2">0,06/100км</td></tr>
</table>

Теперь мы получаем ответ из комбинации просмотра таблицы с вычислениями. Формула для по-минутной платы может быть выражена как определение на псевдоФорте:

```
: ПО-МИНУТНЫЙ-ТАРИФ  ( -- плата-за-минуту)
        ПЛАТА-ЗА-СОЕДИНЕНИЕ  ПЛАТА-ЗА-РАССТОЯНИЕ  + ;
```

Знак "+" теперь появляется один раз в определении, а не девять раз в таблице.

Приняв принцип вычисления как следующий шаг, отметим (или вспомним из первоначального описания проблемы), что оператор просто добавляет одноразовую плату в размере .90 к суммарной. В этом случае плата за оператора не является функцией ни одного из трех измерений. Это наиболее точно может быть выражено в виде "логического расчета"; т.е. функции, комбинирующей логику с арифметикой:

```
: ?ПОМОЩЬ
        ( плата-за-прямой-вызов -- суммарная-плата)
        ОПЕРАТОР?   IF  .90 +  THEN ;
```

(Но помните, такая плата относится только к первой минуте.)

Это дает нам упрощенную таблицу, показанную на рисунке 2-8, и большую степень использования арифметических выражений.

Рис.2-8. Таблица решений без изображения вмешательства оператора.

	Полный тариф		Средний тариф		Низкий тариф	
	1 мин.	Доп. мин.	1 мин.	Доп. мин.	1 мин.	Доп. мин.
Плата за соединение	0,30	0,20	0,22	0,15	0,12	0,09
ПЛЮС						
Плата за расстояние	0,12/100км		0,10/100км		0,06/100км	

Давайте вернемся к нашему определению "ПО-МИНУТНЫЙ-ТАРИФ":

```
: ПО-МИНУТНЫЙ-ТАРИФ  ( -- плата-за-минуту)
        ПЛАТА-ЗА-СОЕДИНЕНИЕ  ПЛАТА-ЗА-РАССТОЯНИЕ  + ;
```

Углубимся в правила вычисления платы за соединение и платы за расстояние.

Плата за соединение зависит от того, первая идет минута или последующая. Поскольку имеется два вида по-минутной оплаты, быть может, было бы проще переписать ПО-МИНУТНЫЙ-ТАРИФ как два разных слова.

Давайте положим, что мы построим компонент, который получает соответствующие величины из таблицы. Слово 1МИНУТА будет давать плату за первую минуту; +МИНУТЫ - за каждую дополнительную. Работа обоих слов будет зависеть от времени дня для получения полного, среднего или низкого тарифа.

Теперь мы можем определить пару слов для замены одного слова ПО-МИНУТНЫЙ-ТАРИФ:

```
: ПЕРВАЯ  ( -- плата)
        1МИНУТА  ?ПОМОЩЬ  ПЛАТА-ЗА-РАССТОЯНИЕ  + ;
: ЗА-ДОПОЛНИТЕЛЬНУЮ  ( -- плата)
        +МИНУТЫ  ПЛАТА-ЗА-РАССТОЯНИЕ  + ;
```

Каково правило для платы за расстояние? Оно очень простое. Это плата за сотню миль, помноженная на расстояние (в сотнях миль). Договоримся, что мы можем составить слово ПЛАТА-ЗА-100МИЛЬ, которое будет выдавать эту величину из таблицы:

```
: ПЛАТА-ЗА-РАССТОЯНИЕ  ( -- плата)
        #МИЛИ @  ПЛАТА-ЗА-100МИЛЬ * ;
```

Наконец, если нам известно общее число минут в вызове, можно вычислить общую плату за вызов:

```
: СУММА  ( -- суммарная-плата)
        ПЕРВАЯ                        ( плата за 1-ю минуту)
        ( #минут) 1-                  ( дополнительные минуты)
        ЗА-ДОПОЛНИТЕЛЬНУЮ *           ( умножить на их стоимость)
        +  ;                          ( сложить вместе)
```

Мы выразили правила для данной задачи через комбинацию простых таблиц и логических вычислений.

(Некоторые замечания в конце этого примера: мы написали нечто, очень похожее на действительную программу на Форте. Но это - всего-навсего псевдокод. Мы остерегались манипуляций со стеком, предполагая, что величины как-то попадают на стек в тех местах, в которых это показано комментариями. Мы также использовали чрезвычайно длинные имена, поскольку они должны быть удобочитаемы для заказчика. В реальной программе предпочтительны короткие имена - см. главу 5.)

Мы развернем законченную программу для этого примера в главе 8.

ОПРЕДЕЛЕНИЕ СТРУКТУР ДАННЫХ

По завершению определения интерфейсов, а иногда и правил, порой возникает необходимость также определить и некоторые структуры данных. Мы будем рассматривать не здесь вопросы реализации таких структур, но лишь описание их концептуальной модели.

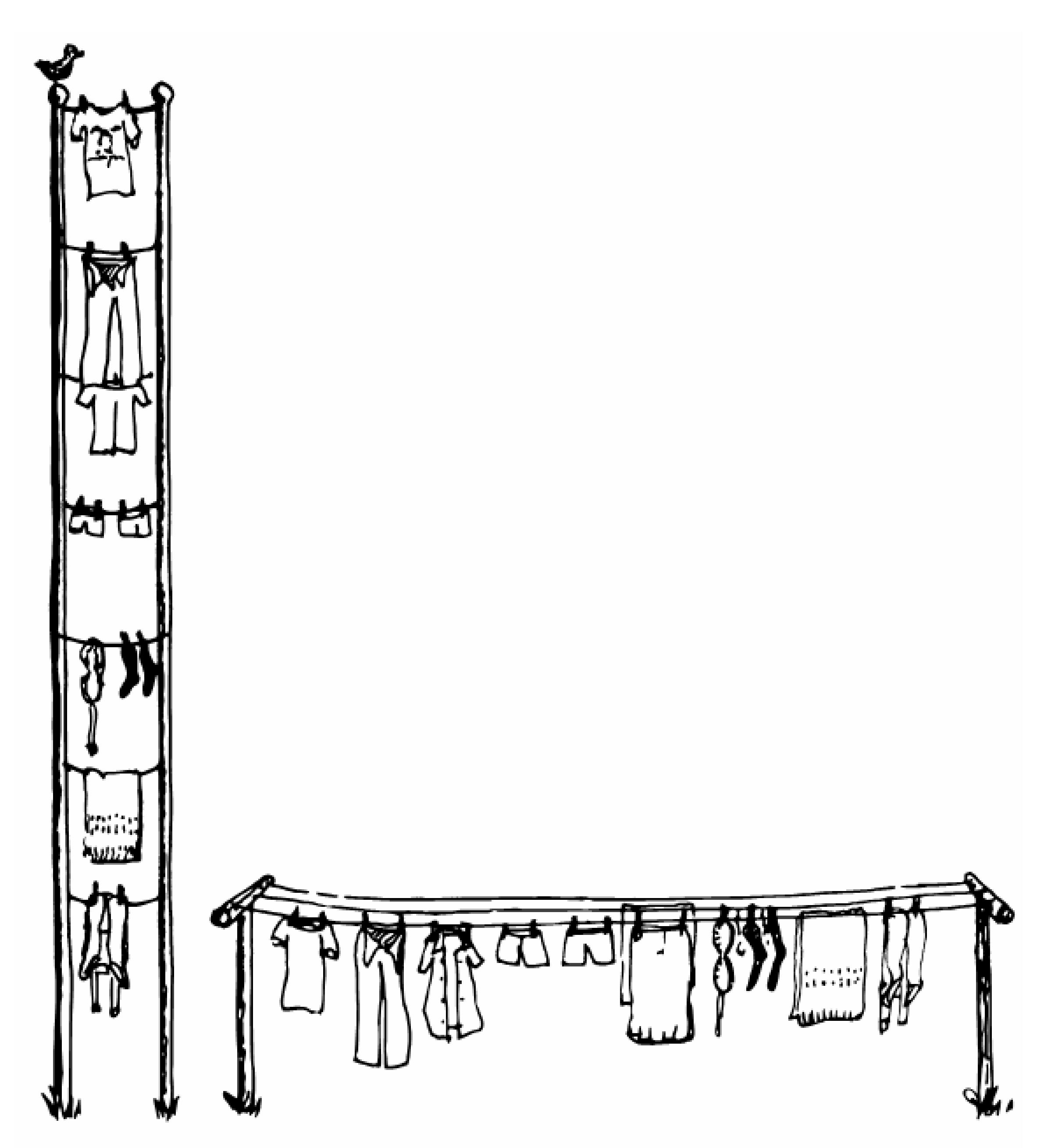
Если, например, Вы автоматизируете библиотечную картотеку, то основная часть Ваших усилий будет касаться разработки логической структуры данных. Вам нужно решить, какую информацию следует хранить для каждой книги: название, имя автора, тему и т.д. Эти "атрибуты" будут составлять "сущность" (набор связанных записей) под названием КНИГИ. Далее Вам надо уточнить, какие другие структуры данных потребуются пользователям для эффективного поиска в списке КНИГИ. Вам может понадобиться другая сущность, состоящая из имен авторов в алфавитном порядке вместе с "указателями атрибутов" тех книг, которые написал каждый автор.

Повлияют на концептуальную модель структуры данных также некоторые ограничения. В примере с библиотечным каталогом Вам нужно знать не только `какая` информация интересует пользователей, но и как долго они расчитывают `ожидать` ее.

Например, пользователи могут запросить список тем по годам их публикаций - скажем, все по дамской галантерее между 1900 и 1910 годом. Если они хотят получать информацию молниеносно, то Вам следует проводить список по годам и темам. Если они могут ждать сутки, Вы можете позволить компьютеру перерыть все подряд книги в библиотеке.

ДОСТИЖЕНИЕ ПРОСТОТЫ

СОВЕТ
Пусть будет просто.



Из имеющихся двух верных решений одно корректируется легче

Во время первых решающих шагов к пониманию проблемы не забывайте про старое высказывание:

Если дано два решения проблемы, то правильное – более простое.

Это особенно верно для разработки программ. Более простое решение часто труднее найти, но если это случилось, то оно:

- легче для понимания
- легче для реализации
- легче для изменения и отладки
- легче для поддержки
- более компактно
- более эффективно
- доставляет больше удовольствия

Одним из наиболее рьяных защитников простоты является Мур:

> Надо иметь чувство размера задачи. Как много потребуется кода для ее реализации? Один блок? Три? Я думаю, что это очень полезный инструмент для разработки. У Вас должно быть внутреннее чувство того, тривиальная или важная это задача, сколько времени и сил Вы можете на нее потратить.
>
> Когда Вы закончили, оглянитесь и скажите: "Обоснованно ли я подошел к решению этой проблемы?" Если Ваше решение занимает шесть экранов, может оказаться, что Вы использовали кувалду для того, чтобы убить комара. Ваш мысленный образ непропорционален важности задачи.
>
> Я видел программы физиков-ядерщиков с сотнями тысяч строк на Фортране. Что бы ни делал этот код, это не оправдывает сотни тысяч строк кода. Видимо, их авторы чрезмерно обобщили задачу. Они решили большую задачу, подмножество в которой составляют их собственные нужды. Они пренебрегли принципом того, что решение должно отвечать задаче.

> --
> СОВЕТ
> Обобщенность обычно приводит к сложности. Не обобщайте свое решение более, чем это необходимо; вместо этого предусматривайте возможность изменения.
> --

Мур продолжает:

> Если перед Вами поставлена задача, Вы можете написать ее решение. После того, как Вы это сделали и обнаружили в нем некоторые неприятности, можете вернуться назад и изменить задачу, получив в конце концов более простое решение.
>
> Существует такой вид оптимизации устройств - минимизация числа вентилей в схеме, где можно использовать преимущества "безразличных" ситуаций. Они возникают или вследствие того, что данный случай на практике не встретится, или потому что действительно все равно. Однако исходные данные зачастую пишутся людьми, ничего не понимающими в программировании. Разработчик может аккуратно описать все случаи, но при этом не сказать Вам, программисту, какие из них действительно важны. Если Вы вольны вернуться

назад и поспорить с ним и использовать преимущества "безразличных" случаев, то можете получить более простое решение.

Возьмем инженерное приложение, такое, как 75-тонный пресс, работающий с металлическим порошком и выдавливающий разные штуки. Некто желает установить компьютер для управления клапанами в тех местах, где сейчас применяется гидравлическое управление. Какого типа спецификацию получите Вы от инженера? Наиболее вероятно, что датчики были расположены с точки зрения электомеханических удобств.

Теперь же их могли расположить где-нибудь еще, но где - инженер позабыл. Если Вам хочется получить объяснения, то Вы должны приблизиться к реальному миру и удалиться от их модели этого мира.

Другим примером может послужить алгоритм ПИД (пропорционального интегрирования и дифференцирования) для сервоприводов. У Вас есть один термин для интегрирования, один - для дифференцирования и один - для сглаживания. Вы комбинируете 30% интегрирования с 10%-ми дифференцирования или что-то вроде этого. Но ведь это - всего-навсего цифровой фильтр. Он был удобен во времена аналоговой техники для того, чтобы можно было отбросить некоторые выражения из области цифровой фильтрации и заявить: "Это - интегратор, а то - дифференциатор. Я сделаю это с помощью конденсатора, а то - с помощью катушки индуктивности". И вновь авторы спецификаций будут моделировать аналоговое решение проблемы, которое моделировало электромеханическое решение и будут находиться за несколько моделей от реальности. На самом деле Вы можете заменить все это с помощью двух или трех коэффициентов в цифровом фильтре для получения куда более чистого, простого и эффективного решения.

СОВЕТ

Вернитесь к тому, что представляла собой задача до того,
как заказчик пытался ее решить. Используйте "безразличные"
ситуации.

Мур продолжает:

Иногда возможности для упрощения сразу не видны. Вот, кстати, проблема увеличения изображения на цифровом графическом дисплее, к примеру, для САПР. На экране имеется картинка, и Вы хотите увеличить ее кусок для рассматривания деталей. Я реализовывал это так: Вы подводили курсор к интересующей позиции, затем нажимали кнопку, и изображение увеличивалось до тех пор, пока у Вас не образовывалось окно желаемого размера. Так я делал всегда. До тех пор, пока не осознал, что это глупо. Мне никогда не нужно было менять увеличение с таким мелким шагом. Поэтому вместо предвижения курсора на один пиксел за раз я сделал перепрыгивание сразу, скажем, на десять позиций. И вместо плавного увеличения размера окна я сделал скачкообразное увеличение. У Вас нет больше выбора масштаба. Увеличение производится в четыре раза. Промежуточные размеры не представляют интереса, прыжки же можно делать

сколько угодно раз. Безжалостно квантуя разные вещи, Вы можете облегчить работу с ними, сделать их более надежными и простыми.

СОВЕТ

Для упрощения - квантуйте.

Мур подводит итоги:

Это большая смелость - взять и сказать: "Вы на самом деле не то имели в виду" или "Вы не будете возражать, если я изыму эту страницу и заменю ее таким вот выражением?" Это вызывает раздражение. От Вас хотят, чтобы Вы делали то, что Вам сказано.

ЛяФарр Стюарт пошел этим путем, когда перепроектировал Форт [3]. Ему не нравился буфер входного потока, поэтому он сделал Форт без него и понял, что буфер ему не нужен.

Если Вы можете улучшить постановку задачи, следует не упускать такую возможность. Гораздо приятнее перепроектировать мир, нежели реализовывать его "в лоб".



An overgeneralized solution.

Опытные программисты учат быть тактичными и проводить свой подход так, чтобы не чувствовалось угрозы: "Какими могли бы быть последствия от замены этого на то?" и т.д. Еще один путь к упрощению проблемы:

СОВЕТ
Упрощая, не доставляйте беспокойства пользователям.

Предположим, Вы проектируете часть текстового процессора, которая показывает каталог хранимых документов на экране, один документ в строке. Задумано, что пользователь будет подводить курсор к имени документа, а затем вводить однобуквенную команду для выбранного действия - "П" для печати, "Р" для редактирования и т.д.

Вначале кажется приемлемым позволить пользователю перемещать курсор по всему экрану. Это означает, что те места, где уже имеется текст, должны быть защищены от порчи. Это приводит к концепции "защищенных полей" и специальной обработки. Более простой подход привязывает курсор к определенным полям, возможно с использованием инверсии цвета для того, чтобы выделить для пользователя величину доступного поля.

Другой случай возникает, когда задача предлагает пользователю ввести числовое значение. Зачастую такие задачи не делают проверку ввода до тех пор, пока Вы не нажмете <ВВОД>, после чего система ответит сообщением об ошибке типа "неверное число". Столь же просто - скорее всего, проще – проверять каждую нажатую клавишу и просто не позволять появляться нецифровым символам.

СОВЕТ
Для упрощения извлекайте выгоду из знания того, что возможно.

Майкл ЛаМанна, Форт-программист из Лонг-Айленда, штат Нью-Йорк, комментирует:

Я всегда пытаюсь проектировать задачу на самом мощном процессоре, на который могу наложить руку. Если у меня есть выбор между разработкой на системе с процессором 68000 либо с процессором 6809, то я буду делать ее на первой системе. Процессор сам по себе настолько мощен, что берет на себя заботу о множестве деталей, которые иначе мне пришлось бы решать самому.

Если мне затем придется вернуться и переписать задачу для более простого процессора, то это нормально. По крайней мере, я не терял своего времени напрасно.

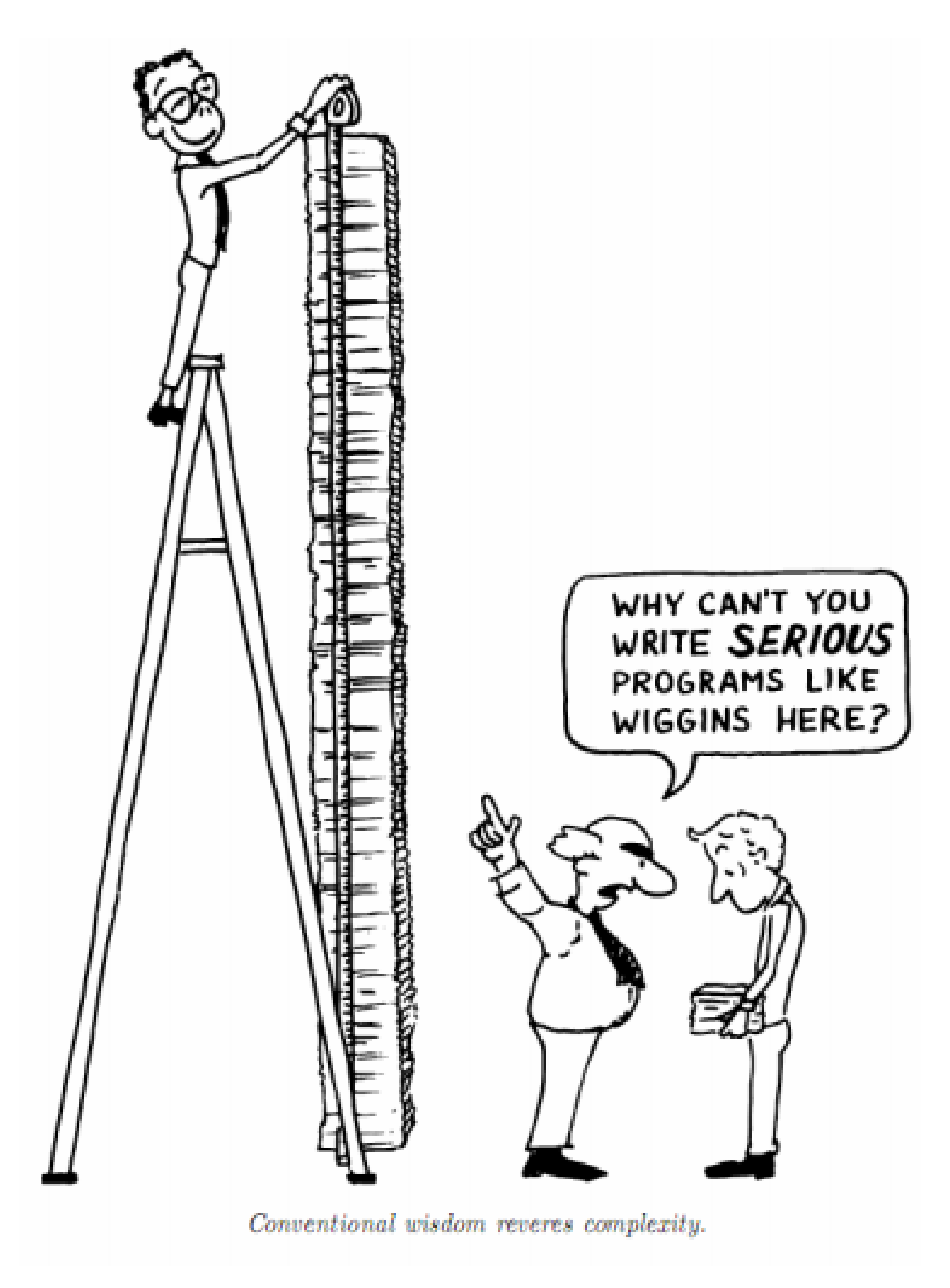
Небольшое предупреждение: Если Вы используете существующий компонент для упрощения своего прототипа, не позволяйте этому компоненту влиять на Ваш проект. Нежелательно, чтобы проект зависел от внутреннего устройства компонента.

СОБЛЮДЕНИЕ БЮДЖЕТА И ГРАФИКА

Другим важным аспектом фазы анализа является распределение стоимости. Опять же, этот процесс сложнее, чем может показаться. Если Вы не знаете проблему до тех пор, пока ее не решите, то как, спрашивается, вы можете узнать, сколько времени она отнимет?

Тщательное планирование обязательно, поскольку все всегда длится дольше, чем Вы расчитываете. У меня по этому поводу есть теория, основанная на законах вероятности:

СОВЕТ
Среднее время доработки задачи "за пару часов" составляет около 12-ти часов.



Conventional wisdom reveres complexity.

Представьте себе следующий сценарий: Вы находитесь посередине разработки большой программы, когда неожиданно Вас осеняет добавить туда относительно маленькую функцию. Вы думаете, что это займет часа два, поэтому, без дальнейшего планирования, Вы ее делаете. Полагаем: два часа на время кодирования. Время проектирования Вы не считаете, поскольку постигли необходимость - и реализацию - функции в мгновение ока при работе над задачей. Поэтому Вы кладете два часа.

Но допустим следующие возможности:

1. Ваша реализация содержит ошибку. Через два часа это не работает. Поэтому Вы проводите еще два часа на переделку. (Всего 4 часа.)
2. ИЛИ еще до реализации Вы поняли, что Ваш начальный проект не работал бы. Вы тратите два часа на перепроектирование. \`Эти\` два часа считаются. Плюс два часа на кодирование. (Всего 4 часа.)
3. ИЛИ Вы реализуете первый проект до того, как понимаете, что он неработоспособен. Поэтому Вы перепроектируете (еще два часа) и перекодируете (еще два). (Всего 6 часов.)
4. ИЛИ Вы кодируете первый вариант, находите ошибку, переписываете код, находите слабость в проекте, перепроектируете, перекодируете, находите ошибку в новом коде, перекодируете опять. (Всего 10 часов.)

Вы видите, как растет снежный ком?

5. Теперь Вам надо задокументировать новую функцию. Добавьте два часа к вышеуказанному. (Всего 12 часов.)
6. После того, как Вы потратитли что-то между 2-мя и 12-ю часами, устанавливая и отлаживая свою новую функцию, Вы неожиданно обнаруживаете, что элемент Y в Вашей задаче перестал работать. Хуже всего то, что Вы не понимаете, почему. Вы проводите два часа за чтением дампов памяти, пытаясь выяснить причину. Когда она найдена, Вы тратите целых 12 дополнительных часов, переделывая элемент Y. (Всего 26 часов.) Затем Вам надо задокументировать синтаксические изменения по элементу Y. (Всего 27 часов.)

Это в сумме составляет более трех человеко-дней. Конечно, редко бывает так, чтобы все эти несчастья навалились на Вас сразу, однако все решительно \`против\` того, чтобы быть таким легким, как кажется.

Как можно улучшить свои шансы по правильной оценке временных запросов? На эту тему написано много хороших книг, отмечу \`Мифический человеко-месяц\` Фредерика П. Брукса, Мл. [4]. У меня есть мало что добавить к этому кладезю знаний, за исключением некоторых личных наблюдений.

1. Не занимайтесь целым. Разбейте проблему на минимально возможные куски, затем посчитайте время на каждый кусок. Сумма частей всегда больше того, что Вы дали бы на целое. (Целое кажется меньшим, чем сумма его частей.)
2. При определении кусков отделяйте те из них, которые Вы настолько хорошо понимаете, что можете рискнуть угадать их содержимое, от тех, для которых этого сделать нельзя. Для второй категории обрисуйте пределы затрат.
3. Немного психологии: всегда предлагайте Вашему клиенту разные варианты. Клиенты \`любят\` варианты. Если вы скажете:
   "Это обойдется Вам в 6000 долларов", клиент наверное ответит: "Я реально потратил бы 4000". Это ставит Вас в такое положение, при котором надо либо соглашаться, либо оставаться без работы.
   Однако, если Вы скажете: "У Вас есть выбор: за 4000 долларов я заставлю это

`проходить` через обруч. За 6000 я заставлю это `прыгать` через обруч. За 8000 я сделаю так, что оно будет `танцевать` через обруч с флагами, разбрасывая конфетти и распевая "Шумел камыш".
Большинство заказчиков сойдутся на прыжках через обруч.

СОВЕТ
Все отнимает больше времени, чем Вы думаете, включая размышления.

СМОТРИНЫ ДЛЯ КОНЦЕПТУАЛЬНОЙ МОДЕЛИ

Последний прямоугольник на нашей итеративной аналитической карусели обозначен как "Демонстрация модели заказчику". С помощью выделенных в этой главе инструментов эта работа должна быть легко выполнима.

При документировании требований помните, что спецификации сходны со снежным человеком. Ныне они могут быть заморожены, но будут двигаться, скользить и растаивать когда пригреет.

Избрали ли Вы диаграммы потоков данных или прямой Форт-псевдокод, готовьтесь к великой оттепели, не забывая применять концепции ограниченной связности.

Покажите задокументированную модель покупателю. Когда заказчик, наконец, удовлетворен, Вы готовы к следующему большому шагу: созданию проекта!

ЛИТЕРАТУРА

1. Kim Harris, "The FORTH Philosophy," `Dr. Dobb's Journal`, Vol. 6, Iss. 9, No. 59 (Sept. 81), pp. 6-11.
2. Victor Weinberg, `Structured Analysis`, Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1980.
3. LaFarr Stuart, "LaFORTH", 1980 FORML Proceedings, p. 78.
4. Frederick P. Brooks, Jr., `The Mythical Man-Month`, Reading, Massachusetts, Addison-Wesley, 1975.

ГЛАВА 3

ПРЕДВАРИТЕЛЬНЫЙ

ПРОЕКТ / ДЕКОМПОЗИЦИЯ

Поскольку Вы имеете некоторое понятие о том, что должна выполнять Ваша программа, пришла пора начинать проектирование. Первый этап - предварительный проект - сосредоточивается на расчленении задачи на обозримые составляющие.

В этой главе мы обсудим два пути декомпозиции Вашей программы на Форте.

ДЕКОМПОЗИЦИЯ ПО КОМПОНЕНТАМ

С Вами когда-нибудь такое случалось? Вы три месяца планировали отправиться на выходные в поход в горы. Вы сочиняли списки того, что нужно взять с собой и грезили о косогорах.

В то же время Вы решали, что надеть на свадьбу Вашей сестры в следующую субботу. У них будет неформальный стиль, и Вы не хотите выбиваться. Все же свадьба есть свадьба. Может быть, Вам все же стоит взять напрокат смокинг.

Несмотря на все эти планы, лишь только в четверг Вы осознали, что эти два события совпадают. В таких случаях хочется как следует выругаться.

Как такой мысленный ляпсус мог случиться с таким умным человеком, как Вы? Кажется, человеческий мозг в действительности устанавливает связи между воспоминаниями. Новые идеи как-то накладываются на существующие пути родственных мыслей.

Figure 3.1: *Pools of thought not yet linked*



В описанном только что несчастном случае не было сделано никакого соединения между двумя отдельно-связанными областями мысли до четверга. Конфликт, по-видимому, возник, когда некоторое новое входное воздействие (что-нибудь тривиальное типа услышанного прогноза погоды на субботу) оказалось связанным одновременно с обеими областями мысли. Молниеностная вспышка осознания прошла между областями, безжалостно преследуемая громоподобной паникой.

Был изобретен простой инструмент для избежания подобных оказий. Он называется календарем. Если бы Вам было нужно записать оба плана на одном его листке, Вы бы увидели отметку о другом плане, то есть то, что Ваш мозг со всем своим запутанным великолепием сделать не смог.

СОВЕТ

Чтобы увидеть связь между двумя вещами, поставьте их рядом вместе. Чтобы напоминать себе об этой связи, `держите` их рядом вместе.

Эти трюизмы подходят к разработке программного обеспечения, особенно для фазы предварительного проектирования. На этой фазе традиционно проектировщик рассекает большую задачу на маленькие, доступные программисту модули.

В первой главе мы обнаружили, что задачи могут быть удобно разбиты на компоненты.

СОВЕТ

Целью предварительного проекта является определение того, какие компоненты необходимы для выполнения поставленных условий.

К примеру, у Вас может быть задача, в которой события должны происходить в соответствии с заранее определенным расписанием. Для соблюдения последнего Вам вначале следует спроектировать несколько слов для образования "лексикона для составления расписаний". С их помощью Вы будете в состоянии описывать порядок событий, который должен соблюдаться в поставленной задаче.

Так с помощью единственного компонента Вы не только разделяете информацию, но также устраняете потенциальные конфликты. Было бы неправильно позволять каждому функциональному модулю "знать" что-либо о его месте в расписании, поскольку это может потенциально привести к конфликтам с расписаниями внутри других модулей.

Как можно узнать при проектировании компонента, какие команды могут понадобиться компонентам-пользователям? Известно, что это нечто вроде проблемы "цыпленка и яйца". Однако Форт-программисты справляются с ней тем же способом, что и цыплята с яйцами: итеративно.

Если компонент хорошо спроектирован, его полнота не играет роли. Фактически компоненту нужно лишь быть достаточным для текущей итерации проектирования. Ни один компонент не должен оставаться "черным ящиком" до конца разработки – это недопустимо для поддерживаемых разработок.

В качестве примера представьте себе, что Ваше произведение должно "говорить" с другими машинами во внешнем мире через универсальную микросхему ввода/вывода, являющуюся частью Вашей системы. Этот чип имеет "управляющий регистр" и "регистр данных". В плохо спроектированной задаче куски кода по всей программе будут обращаться к коммуникационной микросхеме простым выполнением инструкции OUT для засылки соответствующего байта в командный регистр. Это делает задачу в целом бессмысленно зависящей от определенной микросхемы - что очень рискованно.

Вместо этого программисты на Форте написали бы компонент для управления чипом ввода/вывода. Эти команды имели бы логические имена и удобный интерфейс (обычно стек Форта) для обеспечения их использования остальной частью задачи.

На любой итерации проектирования Вашего продукта Вы бы реализовывали только те команды, которые нужны Вам в дальнейшем - но не все возможные коды, которые можно посылать в "управляющий регистр". Если позже в проектном цикле вы бы обнаружили необходимость дополнительной команды, скажем той, что изменяет скорость передачи, то такая команда была бы добавлена к лексикону чипа ввода/вывода, а не в код, потребный для установки скорости. И нет никакой платы за внесение изменения, если не считать нескольких минут (самое большее) на редактирование и перекомпилирование.

СОВЕТ

Внутри каждого компонента реализуйте лишь те команды, которые необходимы на данной итерации. (Но не устраняйте возможности для дальнейших добавлений.)

Что происходит внутри компонента - это совершенно его дело. Не обязательно будет плохим стилем, если определения внутри компонента будут разделять избыточную информацию.

К примеру, запись в определенной структуре данных имеет длину в четырнадцать байтов. Одно из определений в компоненте продвигает указатель на 14 байтов для установки на следующую запись; другое определение уменьшает указатель на 14 байтов.

Пока это число 14 остается "секретом" компонента и не может быть использовано еще где-либо, Вам не нужно и определять его как константу. Используйте лишь число 14 в обоих определениях:

```
: +ЗАПИСЬ   14 ЗАПИСЬ# +! ;
: -ЗАПИСЬ  -14 ЗАПИСЬ# +! ;
```

С другой стороны, если это число требуется вне компонента, или если оно используется внутри компонента много раз, то весьма вероятно, что оно будет изменено. Вам следовало бы спрятать его под именем:

```
14 CONSTANT /ЗАПИСЬ
: +ЗАПИСЬ    /ЗАПИСЬ ЗАПИСЬ# +! ;
: -ЗАПИСЬ    /ЗАПИСЬ NEGATE  ЗАПИСЬ# +! ;
```

(Имя /ЗАПИСЬ по соглашению означает "количество байтов на запись".)

ПРИМЕР: КРОШЕЧНЫЙ РЕДАКТОР

Давайте примем разбиение на компоненты за основную цель. Было бы неплохо спроектировать большую задачу прямо в третьей главе, но, увы, нет места, и мы, конечно, вынуждены будем отложить попытку полностью решить такую задачу.

Вместо этого мы возьмем часть большой задачи, которая уже расчленена. Мы будем проектировать компонент путем дальнейшего его разбиения на компонентики.

Представим себе, что мы должны создать свой крошечный редактор, который позволит пользователям менять контекст полей ввода на экране терминала. К примеру, экран может выглядеть таким образом:

Имя участника `Вера Павловна`

Редактор обеспечит для пользователя три режима смены контекста поля ввода:

`Замещение`. Печать обычных символов замещает прежние символы.

`Удаление`. Нажатие комбинации клавиш "CTRL D" удаляет символ, отмеченный курсором и перемещает остальные символы влево.

`Вставка`. Используя комбинацию клавиш "CTRL I" переводим редактор в режим "вставки", где последовательно нажимаемые обычные символы устанавливаются в позицию, отмеченную курсором, сдвигая остальные символы вправо.

Частью концептуальной модели должна также являться обработка ошибок и исключительных ситуаций; например: каковы размеры поля? что происходит в режиме вставки, когда символы пересекут правую границу? и т.д.

Вот и все данное нам описание. Остальное зависит от нас. Давайте попытаемся определить, какие компоненты нам понадобятся. Во-первых, редактор должен реагировать на нажимаемые клавиши. Поэтому нам понадобится интерпретатор нажатий на клавиши - некая программа, которая при нажатиях ищет соответствие клавишам в списке возможных операций. Такой интерпретатор являет собой один компонент и его лексикон состоит из единственного слова. Поскольку такое слово должно позволять редактирование поля, давайте назовем его РЕДАКТОР.

Операции, вызываемые интерпретатором клавиш, составят второй лексикон. Определения этого лексикона будут выполнять различные требуемые функции. Одно из них может быть названо "СТИРАТЬ", другое "ВСТАВИТЬ" и т.д. Поскольку все эти команды будут вызываться интерпретатором, каждая из них должна обрабатывать одно нажатие клавиши.

Под этими командами должен находиться третий компонент - набор слов, которые реализуют редактируемую структуру данных.

Рис.3-2. Обобщенная декомпозиция задачи создания Крошечного Редактора.

РЕДАКТОР
Строчный интерпретатор
Функции редактирования
Текстовый буфер
Структура данных
Команды

Наконец, нам будет нужен компонент для демонстрации поля на видеоэкране. Во имя простоты давайте запланируем создание всего одного слова ПОКАЗАТЬ для смены изображения всего поля после каждого нажатия на клавишу.

```
: РЕДАКТОР   BEGIN  KEY ПРОВЕРИТЬ ПОКАЗАТЬ  ... UNTIL ;
```

Этот подход отделяет проверку буфера от регенерации дисплея. Отныне мы сконцентрируемся на проверке буфера.

Давайте рассмотрим каждый компонент по отдельности и попытаемся определить каждое слово, которое нам понадобится. Мы можем начать с рассмотрения событий, которые должны происходить внутри каждой из наиболее важных функций редактора: замещения, стирания и вставки. Мы можем нацарапать нечто вроде нижеследующего на обратной стороне старого ресторанного меню (сейчас не будем обращать особого внимания на обработку исключительных случаев):

`Для Замещения`:	ФУН**Х**ЦИОНАЛЬНОСТЬ
Записывать новый символ в байт, на котором стоит указатель.	ФУН**К**ЦИОНАЛЬНОСТЬ
Продвинуть указатель (если он не в конце поля).	ФУНКЦИОНАЛЬНОСТЬ
`Для Стирания`:	ФУНКЦИОН**С**АЛЬНОСТЬ
Скопировать на одну позицию влево строку, начинающуюся справа от указателя.	ФУНКЦИОНАЛЬНОСТЬЬ
Записать "пробел" в последнюю позицию в строке.	ФУНКЦИОНАЛЬНОСТЬ_
`Для Вставки`:	ФУ**К**ЦИОНАЛЬНОСТЬ
Скопировать вправо на одну позицию строку, начинающуюся от указателя.	ФУККЦИОНАЛЬНОСТЬ
Записать новый символ в байт, на котором установлен указатель.	ФУНКЦИОНАЛЬНОСТЬ
Продвинуть указатель (если не конец поля).	ФУНКЦИОНАЛЬНОСТЬ

Мы только что "на одной ноге" разработали алгоритмы для задачи.

Наш следующий шаг состоит в исследовании этих глвных процедур для поиска полезных "имен" - процедур или элементов, которые могут быть:

1. возможно, использованными вторично, либо
2. возможно, измененными

Мы поняли, что все три процедуры используют нечто, называемое "указателем". Нам нужно две процедуры:

1. для получения значения указателя (если его отсчет относителен, такая функция будет произволить некоторые расчеты).
2. для продвижения указателя.

Постойте, три процедуры:

3. для перемещения указателя назад.

поскольку мы захотим, чтобы "клавиши управления курсором" перемещали его вперед и назад без редактирования.

Все три эти оператора будут ссылаться на физический указатель где-то в памяти. Как и где он будет храниться (относительно или абсолютно) должно быть спрятано внутри компонента.

Давайте сделаем попытку переписать эти алгоритмы в коде:

```
: КЛАВИША# ( дает код последней нажатой клавиши) ... ;
: ПОЗИЦИЯ ( дает адрес символа по указателю) ... ;
: ВПЕРЕД ( продвигает указатель, остановка в конце) ... ;
: НАЗАД ( уменьшает указатель, остановка в начале) ... ;
: ЗАМЕСТИТЬ    КЛАВИША# ПОЗИЦИЯ C!  ВПЕРЕД ;
: ВСТАВИТЬ     СМЕСТИТЬ> ЗАМЕСТИТЬ ;
: СТЕРЕТЬ      СМЕСТИТЬ<  ОЧИСТИТЬ-КОНЕЦ ;
```

Для копирования текста налево и направо нам пришлось по мере написания придумать два новых имени - СМЕСТИТЬ< и СМЕСТИТЬ> (произносится "сместить-назад" и "сместить-вперед" соответственно). Оба они, конечно, будут использовать слово ПОЗИЦИЯ, а также должны опираться на элемент, который мы предварительно определили как "знающий" длину поля. Мы можем приняться за это, когда доберемся до написания третьего компонента.

Но посмотрите, что мы уже обнаружили: можно описать "Вставку" как просто "СМЕСТИТЬ> ЗАМЕСТИТЬ".

Другими словами, "Вставка" в действительности `использует` "Замещение" несмотря на то, что они кажутся существующими на одинаковом уровне (по крайней мере, с точки зрения Структурированного Программиста).

Вместо углубления в третий компонент, давайте изложим наши знания о первом компоненте, интерпретаторе клавиш. Во-первых, мы должны разрешить проблему "режима вставки". При этом выясняется, что "вставка" - это не просто нечто, случающееся когда Вы нажимаете на определенную клавишу, как в режиме стирания. Это `другой способ интерпретации` некоторых из возможных нажатий на клавиши.

К примеру, в режиме "замещения" обычный символ записывается в текущую позицию курсора; но в режиме "вставки" остальная часть строки должна быть сдвинута вправо. И клавиша забоя также работает по-другому, когда редактор находится в режиме вставки.

Поскольку имеются два режима, "вставки" и "не-вставки", интерпретатор клавиш должен присваивать клавишам два возможных набора именованных процедур.

Мы можем записать наш интерпретатор нажатий на клавиши в виде таблицы решений (позаботясь о реализации позднее):

`Клавиша`	`Не-вставка`	`Вставка`
Ctrl-D	СТЕРЕТЬ	ВЫКЛ-ВСТАВКУ
Ctrl-I	ВКЛ-ВСТАВКУ	ВЫКЛ-ВСТАВКУ
забой	НАЗАД	НАЗАД<
стрелка-влево	НАЗАД	ВЫКЛ-ВСТАВКУ
стрелка-вправо	ВПЕРЕД	ВЫКЛ-ВСТАВКУ
ввод	ВЫХОД	ВЫКЛ-ВСТАВКУ
любой видимый символ	ЗАМЕСТИТЬ	ВСТАВИТЬ

Мы поместили возможные типы клавиш в левой колонке, то, что они делают в нормальном режиме - в средней колонке, а для режима "вставки" - в правой колонке.

Для реализации случая нажатия "забоя" в режиме вставки мы добавили новую процедуру:

```
: НАЗАД<    НАЗАД  СМЕСТИТЬ< ;
```

(передвинуть курсор назад к последнему введенному символу, затем сдвинуть все справа налево, перекрывая ошибку).

Эта таблица кажется наиболее логичным изображением задачи на текущем уровне. Мы оставим реализацию для будущего рассмотрения (глава 8).

Теперь продемонстрируем огромную ценность подобного подхода с точки зрения управляемости. Мы подбросим себе задачу - существенное изменение в планах.

ПОДДЕРЖКА ЗАДАЧИ, ОСНОВАННОЙ НА КОМПОНЕНТАХ

Насколько хорошо наш проект поведет себя перед лицом изменений? Вообразим следующий сценарий:

Вы изначально согласились, что можем обновлять видеодисплей простым переписыванием всего поля всякий раз после нажатия на клавишу. Мы даже реализовали такой код на нашем персональном компьютере с его видеопамятью, входящей в основное адресное пространство; код, обновляющий всю строку за время мерцания развертки экрана. Но теперь заказчик хочет, чтобы задача работала в сети на основе телефонных линий, в которой весь ввод/вывод производится весьма неторопливо. Поскольку некоторые из наших полей ввода занимают почти всю ширину экрана, например, 65 символов, было бы слишком долго обновлять всю строку после каждого нажатия на клавишу.

Нам придется изменить задачу так, чтобы обновлять только ту часть поля, которая действительно меняется. При "вставке" и "стирании" это означало бы текст справа от курсора. При "замещении" это означало бы замену только одного символа.

Такое изменение существенно. Функция регенерации изображения, которую мы по-рыцарски отделили от интерпретатора клавиш, ныне зависит от текущей функции редактирования. Как мы обнаружили, для реализации интерпретатора наиболее важны имена:

```
ВПЕРЕД
НАЗАД
ЗАМЕСТИТЬ
ВСТАВИТЬ
СТЕРЕТЬ
НАЗАД<
```

Ни одно из определений не делает ссылок к процессу обновления изображения, поскольку это изначально предполагалось делать позже.

Но не все так плохо, как кажется. При внимательном взгляде процесс ЗАМЕСТИТЬ мог бы легко включать в себя команду для печати нового символа в позиции курсора. А СМЕСТИТЬ< и СМЕСТИТЬ> могли бы иметь команды для распечатки всего текста справа от этой позиции (включая ее саму), а затем возврата курсора дисплея в его текущее положение.

Вот наши пересмотренные определения:

```
: ЗАМЕСТИТЬ   КЛАВИША# ПОЗИЦИЯ С!  КЛАВИША# EMIT  ВПЕРЕД ;
: РАСПЕЧАТАТЬ  ( напечатать от текущей позиции до конца поля и вернуть
курсор) ... ;
: ВСТАВИТЬ    СМЕСТИТЬ>  РАСПЕЧАТАТЬ  ЗАМЕСТИТЬ ;
: СТЕРЕТЬ     СМЕСТИТЬ<  ОЧИСТИТЬ-КОНЕЦ  РАСПЕЧАТАТЬ ;
```

Поскольку имеются всего три функции, изменяющие память, нужны всего три функции для обновления экрана. Такая идея небесспорна. Мы должны быть способны отстаивать такие утверждения для обеспечения корректности программы.

Заметьте, что дополнительная проблема регенерации изображения принуждает ввести дополнительный "указатель": текущего положения курсора на экране. Однако компонентная декомпозиция вдохновила нас рассматривать процесс ЗАМЕСТИТЬ как изменяющий одновременно поле данных и его видеоизображение; то же самое со СМЕСТИТЬ< и СМЕСТИТЬ>. По этой причине кажется естественным сохранить лишь один реальный указатель - относительного типа - из которого мы можем вычислить либо адрес данных в памяти, либо номер колонки на экране.

Поскольку природа указателя полностью спрятана внутри трех процессов: ПОЗИЦИЯ, ВПЕРЕД и НАЗАД, то мы можем немедленно применить такой подход, несмотря на то, что вначале наш подход был другим.

Такое изменение может показаться слишком простым – даже очевидным. Если это так, то потому, что технология обеспечивает гибкую разработку. Если бы мы использовали традиционный подход - делали проектирование в соответствии со структурой или в соответствии с последовательным процессом преобразования данных - наш хрупкий проект был бы вдребезги разбит такими переменами.

Для доказательства такого утверждения нам придется начать все опять заново.

ПРОЕКТИРОВАНИЕ И ПОДДЕРЖКА ЗАДАЧИ ПРИ ТРАДИЦИОННОМ ПОДХОДЕ

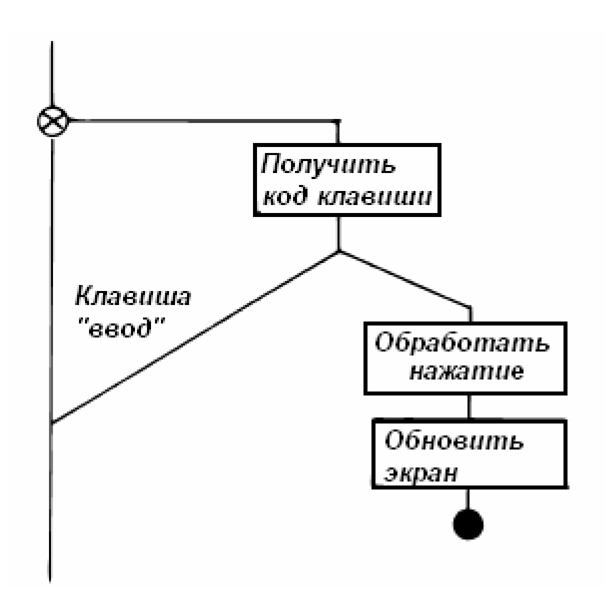
Давайте сделаем вид, что мы пока что не изучали проблему создания Крошечного Редактора и имеем вновь минимальное его описание. Мы также начнем с нашего первого допущения того, что можно обновлять изображение перебивкой всего поля после каждого нажатия на клавишу.

В соответствии с нисходящим методом проектирования давайте окинем проблему возможно более широким взглядом. На рисунке 3-3 наша программа изображена в своих простейших терминах. Здесь мы видим, что редактор на самом деле представляет собой цикл, который продолжает получение нажатий на клавиши и выполнение некоторых функций редактирования до тех пор, пока пользователь не надавит клавишу "ввод".

Внутри цикла у нас имеется три модуля: получения символа с клавиатуры, редактирования данных и, наконец, обновления дисплея на предмет соответствия этим данным.

Ясно, что большая часть работы будет происходить внутри "обработки клавиши".

Рис.3-3. Традиционный подход: взгляд с вершины.



Применение метода последовательной детализации дает показанную на рисунке 3-4 расшифровку задачи "обработка клавиши". Мы обнаружили, что для достижения такой конфигурации потребовалось несколько попыток. Проектирование на этом уровне вынуждает нас учитывать одновременно множество тех вещей, которые мы оставляли на будущее в предыдущей попытке.

Рис.3-4. Структура задачи "Обработка Клавиши".

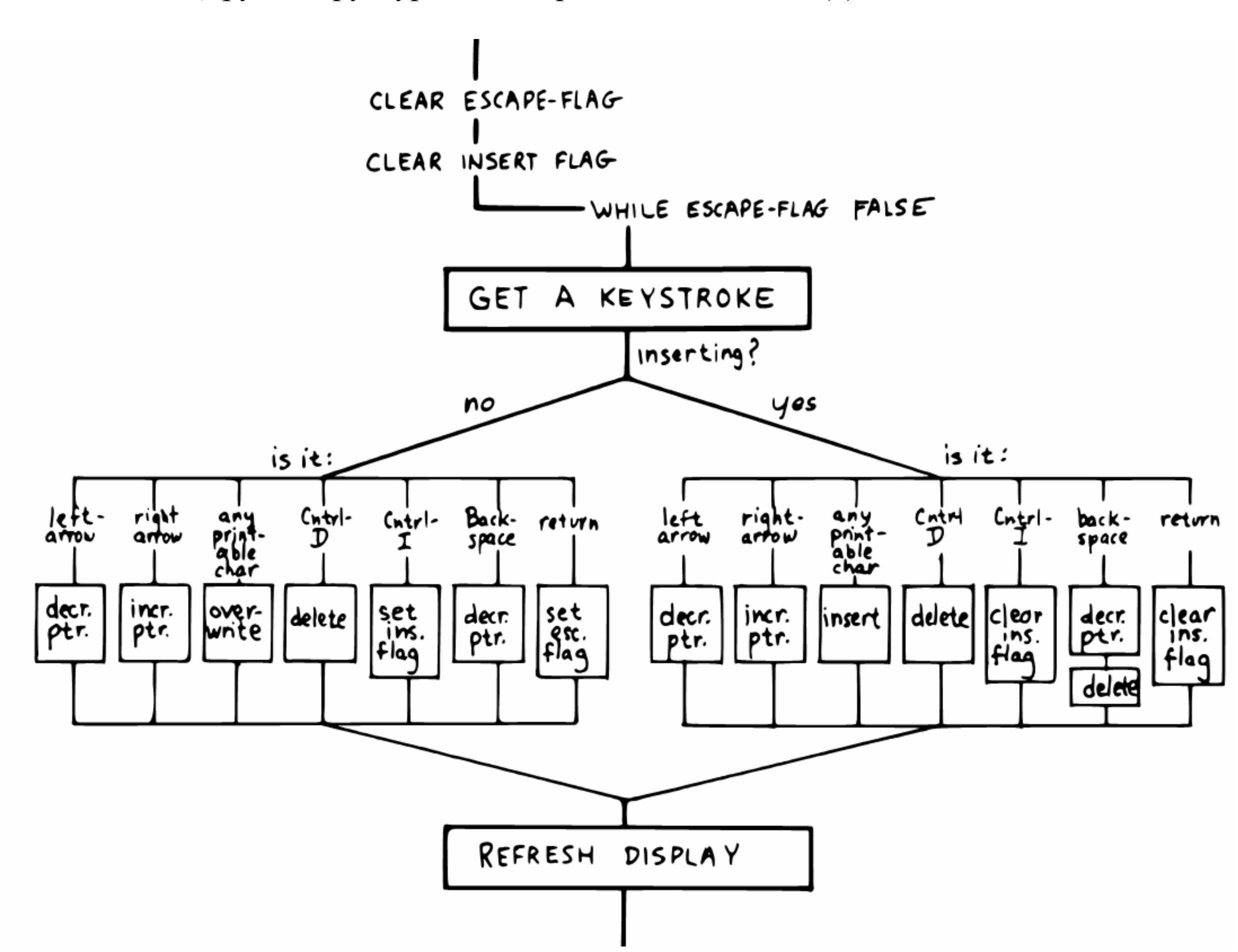
ОЧИСТИТЬ ФЛАГ ВЫХОДА
ОЧИСТИТЬ ФЛАГ ВСТАВКИ
пока флаг выхода = ложь
ПОЛУЧИТЬ КОД КЛАВИШИ
если клавиша
Стрелка влево — Стрелка вправо — Любой печатный символ — Ctrl-D — Ctrl-I — Забой — Ввод
Уменьш. указатель
Увелич. указатель
Вставка? Нет: Замещение; Да: Вставка
Удаление
Вставка? Нет: Устан. флаг вставки; Да: Сброс. флаг вставки
Вставка? Нет: Уменьш. указатель; Да: Уменьш. указатель, Удаление
Устан. флаг выхода
ОБНОВИТЬ ЭКРАН

К примеру, мы должны учитывать все клавиши, которые могут быть нажаты. Что более существенно, нам приходиться принимать во внимание проблему "режима вставки". Такая реализация вынуждает нас вводить ФЛАГ ВСТАВКИ, который изменяется при нажатии на "Ctrl-I". Он используется внутри нескольких линий структуры для определения того, как обрабатывать ту или иную клавишу.

Другой флаг, названный ФЛАГ ВЫХОДА, вроде бы дает хорошую возможность обеспечить структурированный выход из цикла редактирования, если пользователь нажимает клавишу ввода.

К моменту окончания диаграммы нас проверки на режим вставки замучили. Нельзя ли было бы проверять этот режим один раз, в самом начале? Мы делаем в соответствии с этим другой чертеж (рисунок 3-5).

Рис.3-5. Другая структура для "Обработки Клавиши" (*).



Как видно, он оказывается даже еще более ужасным, чем первый. Теперь мы делаем проверку на каждую клавишу по два раза. Хотя, конечно, интересно как, будучи функционально эквивалентными, две структуры оказываются совершенно различными. Одного этого достаточно, чтобы усомниться в том, действительно ли структуры управления так уж сильно связаны с задачей.

Остановясь на первом рисунке, мы в конце концов пришли к наиболее важным модулям - тем, которые делают собственно замещение, вставку и стирание. Еще раз взгляните на нашу расшифровку "Обработки Клавиши" на рисунке 3-4. Давайте остановимся на одной из семи возможных линий процесса выполнения, той, которая возникает при получении видимого символа.

На рисунке 3-6(а) виден исходный структурный путь для видимого символа.

Поскольку мы выделили алгоритмы для замещения и вставки символов, то должны детализировать картину, как показано на рисунке 3-6(б). Но посмотрите на возмутительную избыточность кода (обведено кружочками). Большинство знающих программистов поняли бы ненужность такой избыточности и изменили бы структуру так, как показано на рисунке 3-6(в).

Рис.3-6. Одна и та же часть, "детализированная" и "оптимизированная".

Не так уж плохо, не правда ли?

ИЗМЕНЕНИЕ В ПЛАНАХ.

О'кей, приготовились, и вот - грянули изменения. Нам объявили, что эта задача не будет теперь использоваться на дисплее с прямо доступной видеопамятью. Что сделает это изменение со структурой нашего проекта?

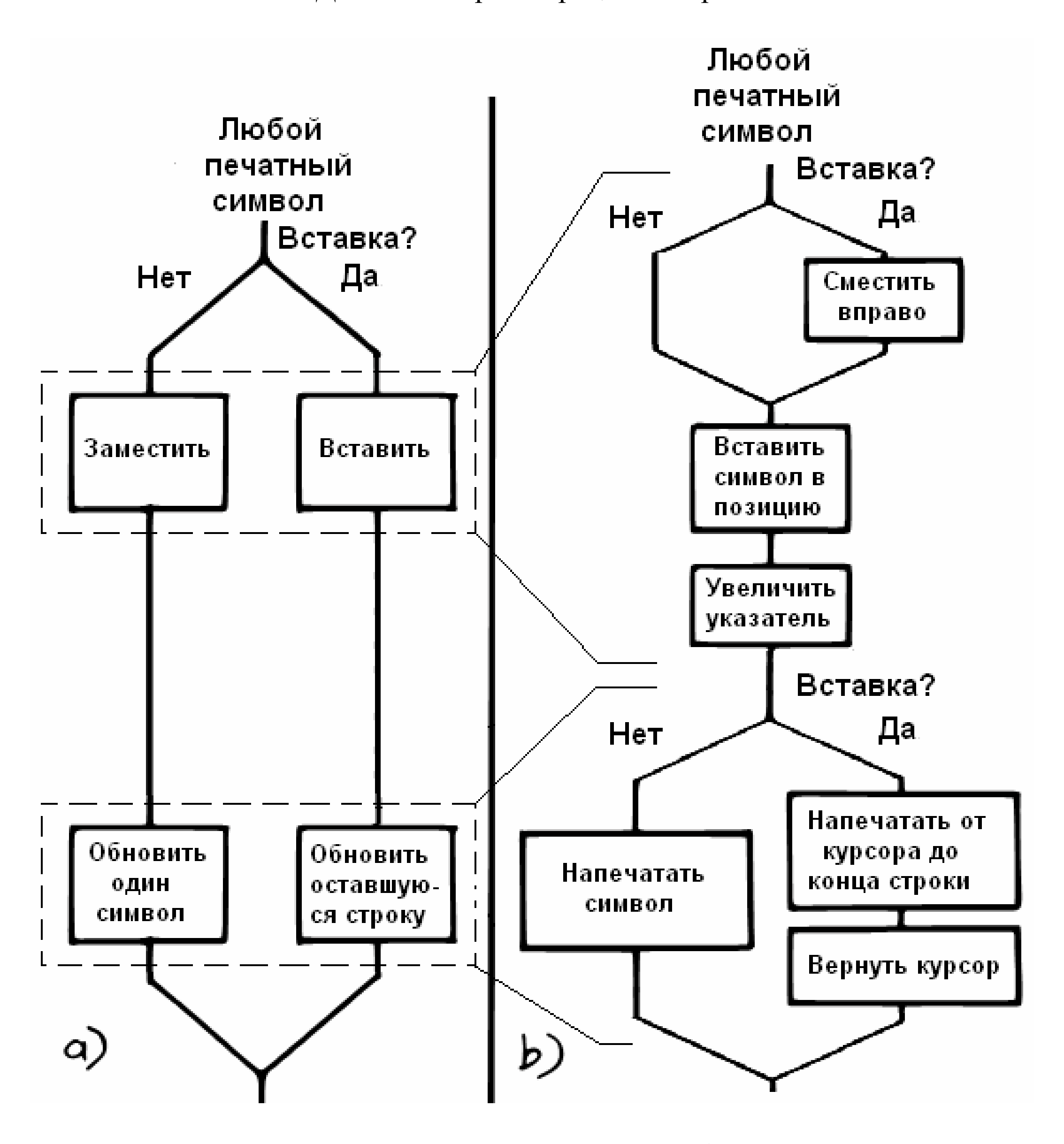
Ну, для начала оно разрушает "Обновление Изображения" как независимый модуль. Функция "обновления изображения" ныне распределена между различными структурными линиями внутри "Обработки Клавиши". Структура нашей задачи в целом изменилась.

Легко увидеть, как мы могли бы неделями производить нисходящее проектирование только ради того, чтобы обнаружить, что спускались не по тому дереву.

Что происходит, когда мы пытаемся изменить программу? Давайте еще раз взглянем на путь прохождения любого видимого символа.

На рисунке 3-7(а) показано, что происходит с нашим первым проектом, когда добавляется регенерация изображения. Часть (б) показывает наш "оптимизированный" проект с развернутыми модулями обновления. Заметьте, что мы проверяем теперь флаг вставки дважды внутри этого единственного ответвления внешнего цикла.

Рис.3-7. Добавление регенерации изображения.



Но, что еще хуже, в нашем проекте есть ошибка. Вы можете ее найти?

В обоих случаях, при замещении и вставке, указатель продвигается `до` регенерации. В случае замещения мы показываем новый символ в неправильном месте. В случае вставки мы перебиваем остаток строки без нового символа.

Допустим, такую ошибку легко отследить. Нам нужно только переместить модули регенерации вверх до "увеличения указателя".

Дело в другом: как мы ее пропустили? А просто мы были заняты потоком управления - поверхностным элементом проектирования программ.

Наоборот, в нашем по-компонентном проекте правильное решение вытекает естественным образом, поскольку мы "использовали" компонент для регенерации внутри

редактирующего компонента. Мы также использовали ЗАМЕЩЕНИЕ внутри слова ВСТАВКА.

Разбивая нашу задачу на компоненты, использующие друг друга, мы достигли не только `элегантности`, но и более прямого пути к `корректности`.

ИНТЕРФЕЙСНЫЙ КОМПОНЕНТ

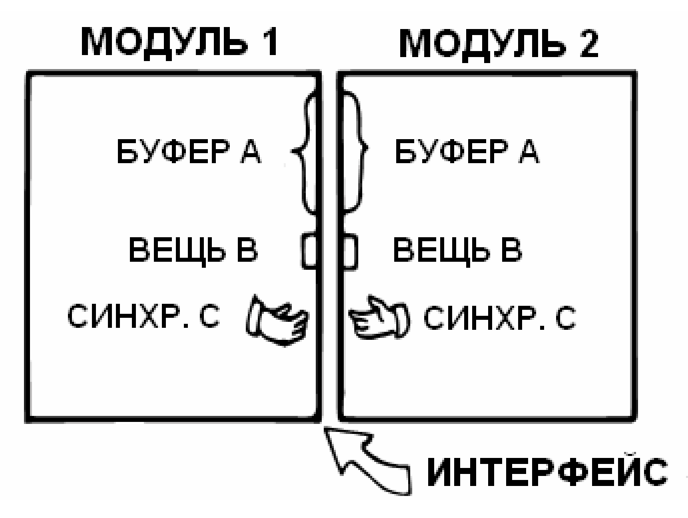
В терминах компьютерной науки взаимодействие между модулями имеет два аспекта. Во-первых, есть способ, по которому другие модули `вызывают` данный; это - управляющий интерфейс.

Во-вторых, есть способ, по которому модули передают и получают данные; это - интерфейс данных.

Благодаря словарной структуре Форта организация управления не представляет трудностей. Определения вызываются просто по именам. Поэтому мы в этом разделе будем использовать слово "интерфейс", имея в виду интерфейс данных.

Когда дело доходит до итерфейсов данных между модулями, традиционная мудрость говорит только о том, что "интерфейсы должны быть тщательно продуманы и минимально сложны". Причина для такой тщательности, конечно, состоит в том, что каждый из модулей должен держать свой конец такого интерфейса (рисунок 3-8).

Рис.3-8. Традиционный взгляд на интерфейс как на соединение.



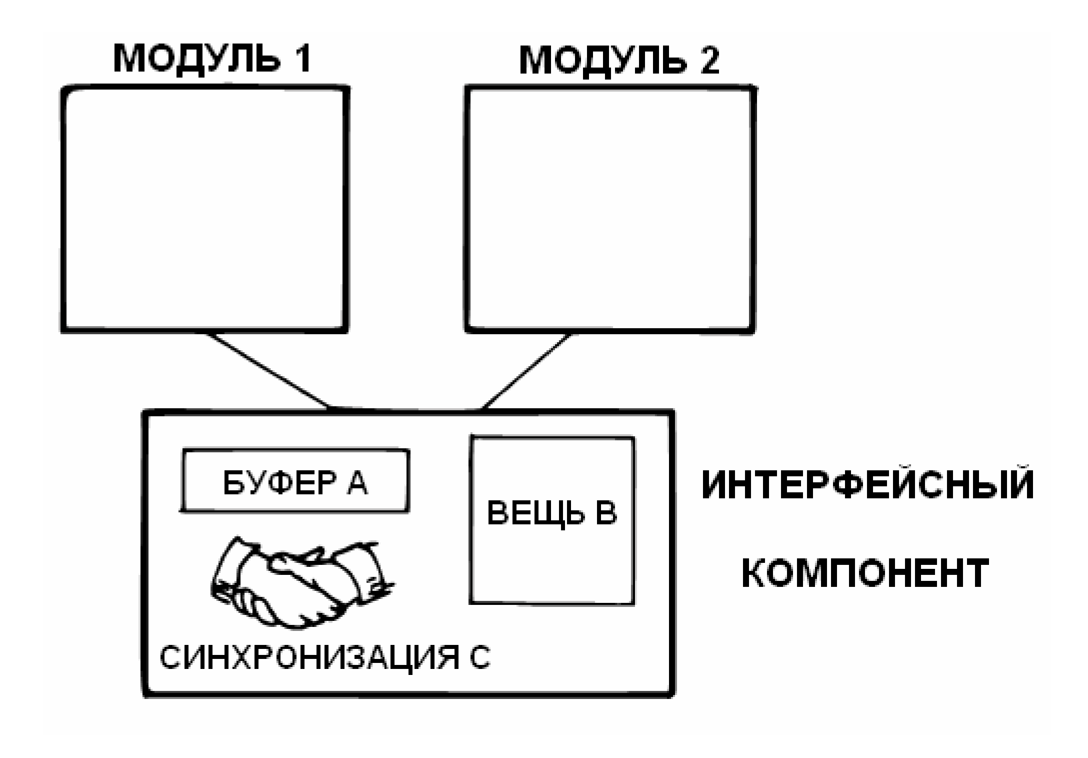
Это предопределяет наличие избыточного кода. Как мы видели, избыточность рождает, по крайней мере, две проблемы: неуклюжий код и плохую управляемость. Изменение интерфейса в одном модуле будет сказываться на другом модуле.

Имеется лучший способ обеспечить интерфейс, нежели приведенный. Позвольте мне предложить проектный элемент, который я называю "интерфейсным компонентом". Целью введения такого компонента является реализация и `упрятывание информации` об интерфейсе данных между двумя (или более) компонентами (рисунок 3-9).

СОВЕТ

Как структуры данных, так и команды, принимающие участие в коммуникациях между модулями, должны быть выделены в интерфейсный компонент.

Рис.3-9. Использование интерфейсного компонента.



Позвольте привести пример из моего недавнего опыта. Одним из моих хобби является создание форматтеров текста/редакторов. (Я их разработал два, включая тот, на котором пишу эту книгу.)

В моей последней разработке часть форматтера имеет два компонента. Первый считывает исходный документ и решает, где сделать перевод строки, где - разделение между страницами и т.д. Но, вместо посылки строки на принтер или терминал, он сохраняет ее на время в "строчном буфере".

Аналогично, вместо посылки команд управления принтером - для включения курсива, подчеркивания и т.п. – при форматировании, он предопределяет эти команды до тех пор, пока текст действительно не будет выдан. Для такого предопределения я завел другой буфер, названный "буфером атрибутов". Он соотносится, байт к байту, с буфером строки, и в каждом из его байтов содержится набор флагов, показывающих, что соответсвующий символ должен быть подчеркнут, сделан курсивом или как-нибудь еще.

Второй компонент показывает или печатает содержимое буфера строки. Компонент знает, выдается ли строка на терминал или на принтер и дает текст в соответствии с признаками, указанными в буфере атрибутов.

Мы имеем здесь два хорошо определенных компонента - формирователь строки и компонент вывода, каждый из которых поддерживает часть функций форматтера в целом.

Интерфейс данных между этими двумя компонентами чрезвычайно сложен. Он состоит из двух буферов, переменной, показывающей текущее число символов в них и, наконец, - "знаний" о том, что означает вся эта система атрибутов.

На Форте я определил эти элементы все вместе на единственном экране. Буферы определены с помощью CREATE, счетчик - обычная переменная VARIABLE, а атрибуты заданы в виде констант (CONSTANT), как, например:

```
1 CONSTANT ПОДЧЕРКИВАНИЕ  ( маска битов для подчеркивания)
2 CONSTANT КУРСИВ  ( маска битов для курсива)
```

Форматирующий компонент использует фразы типа ПОДЧЕРКИВАНИЕ УСТАНОВИТЬ для установки битов в буфере атрибутов. Компонент вывода использует фразы типа ПОДЧЕРКИВАНИЕ AND для анализа буфера атрибутов.

ОШИБКА В ПРОЕКТЕ.

При проектировании интерфейсного компонента следует спросить себя: "каков набор структур и команд, которые должны использоваться совместно сообщающимися компонентами?" Важно определить, какие элементы принадлежат интерфейсу, а какие должны оставаться внутри одного из компонентов.

При написании своего текстового форматтера я не смог полностью ответить на этот вопрос и сделал ошибку. Проблема была в следующем:

Я допустил возможность использования шрифтов различной ширины: уплотненных, с двойной шириной и т.д. Это означает не только посылку различных сигналов в принтер, но также изменение числа символов, допустимых для одной строки.

У меня в форматтере имеется переменная под именем СТЕНА. СТЕНА показывает правую границу: точку, после которой нельзя располагать текст. Применение различных величин ширины означает пропорциональное изменение содержимого переменной СТЕНА. (В действительности это уже само по себе оказывается ошибкой. Мне следовало бы использовать более качественную единицу измерения, величина которой оставалась бы постоянной для строки. Изменение ширины печати означало бы изменение количества таких единиц на один символ. Но подручными средствами исправлять ошибку ...)

Увы, я использовал переменную СТЕНА также внутри компонента вывода для подсчета количества выводимых символов. Я рассчитывал, что эта величина будет меняться в зависимости от того, какую ширину печати я использую.

И я был прав - 99% времени. Но однажды я обнаружил, что при определенных условиях строка из уплотненного текста как-то урезывалась. Последние несколько слов отсутствовали. Причиной оказалось то, что СТЕНА изменялась до того, как у компонента вывода появлялась возможность ее использовать.

В начале я не видел ничего плохого в том, чтобы позволить этому компоненту запросто использовать переменную СТЕНА из форматирующего компонента. Теперь я осознал, что форматер должен был оставлять другую переменную для компонента вывода для указания последнему числа подготовленных символов в буферах. Это не дало бы возможности никаким последующим командам изменения ширины изменить содержимое переменной СТЕНА.

Важно было, чтобы два буфера, команды атрибутов и новая переменная были `единственными` элементами, которые могли совместно использоваться обоими модулями. Доступ внутрь модуля из другого модуля может накликать беду.

Мораль этой истории состоит в том, что необходимо делать различие между структурами данных, которые правильно используются внутри единственного компонента и теми, которые могут быть совместно использованы более чем одним компонентом.

Родственное замечание:

СОВЕТ
Выражайте в реальных единицах любые данные, которые разделяются компонентами.

Для примера:

Модуль А измеряет температуру в печи.
Модуль Б управляет горелкой.
Модуль В контролирует, что дверца закрыта, если печь достаточно горяча.

Информация, интересная всем - это температура печи, выраженная непосредственно в градусах. Хотя модуль А может получать величину, представляющую собой напряжение от термодатчика, он должен преобразовать ее в градусы перед выдачей результата остальной задаче.

РАЗБИЕНИЕ ПО ПОСЛЕДОВАТЕЛЬНЫМ УРОВНЯМ СЛОЖНОСТИ

Мы обсуждали один путь декомпозиции: по компонентам. Другой путь - это путь по последовательным уровням сложности.

Одним из правил Форта является то, что слово для вызова или для ссылки на него должно быть определено заранее. Обычно последовательность, в которой определяются слова, соответствует порядку возрастания функций, которые они должны делать. Такая последовательность приводит к естественной организации исходных текстов. Более мощные команды просто добавляются на вершину элементарных (рисунок 3-10а).

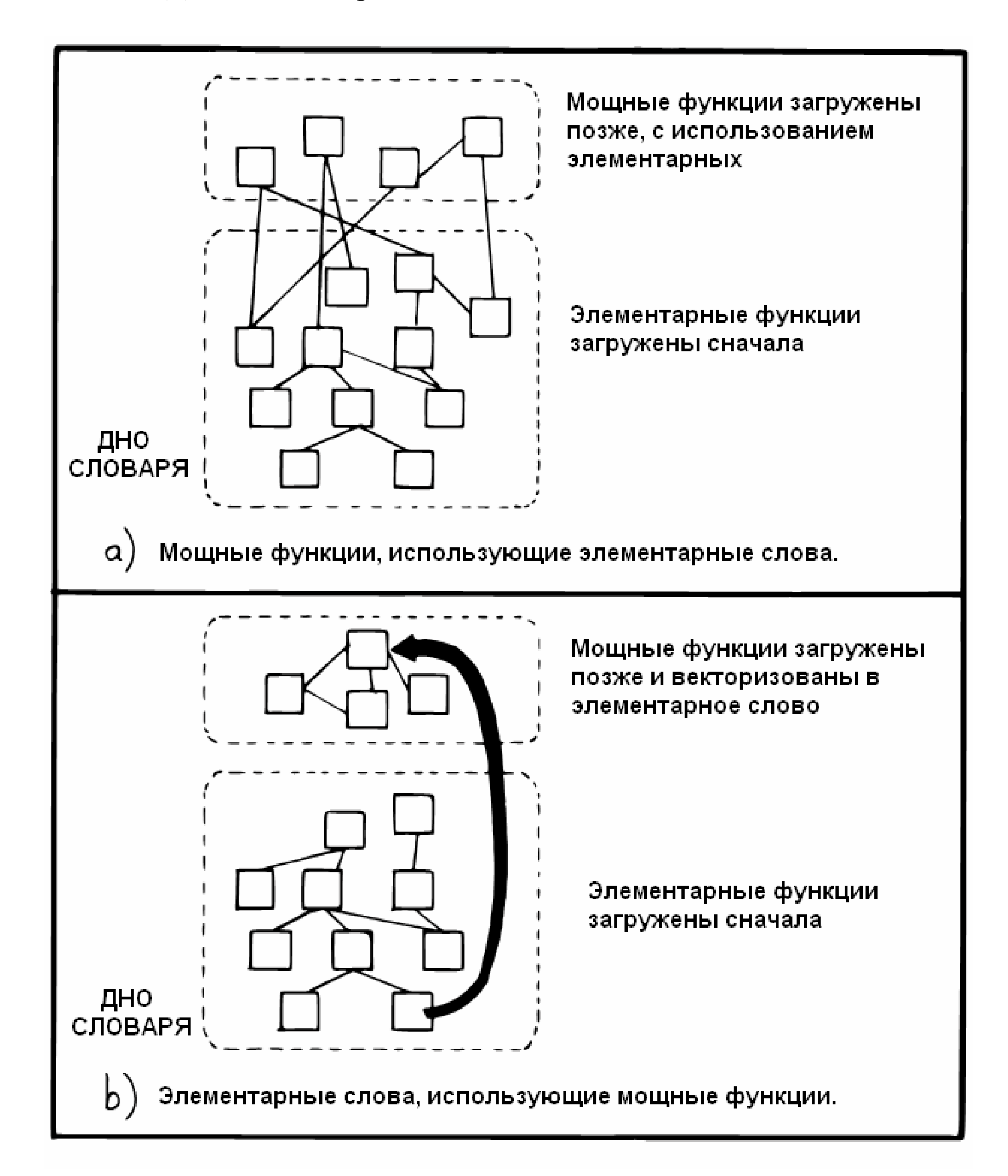
Вначале идут простейшие, как в букваре. Новичок, пробующий разобраться в проекте, имеет возможность прочитать элементарные части кода по мере движения к более углубленным.

Однако во многих крупных задачах дополнительный выигрыш лучше всего достигается путем улучшения некоторых начальных, корневых частей задачи (рисунок 3-10б). Имея возможность изменять работу слов на нижнем уровне, пользователь может менять возможности всех команд, которые используют корневые слова.
Возвращаясь в качестве примера вновь к текстовому процессору, рассмотрим одну из его примитивных функций - ту, которая переводит новую страницу. Она используется словом, переводящим новую строку; она вызывается, когда в странице кончаются строки. В свою очередь слово, переводящее строку, используется тем, которое форматирует слова в строке; когда очередное слово не влезает в строку, вызывается ПЕРЕВОД-СТРОКИ. Такая иерархия "использования" предполагает, что мы определили ПЕРЕВОД-СТРАНИЦЫ раньше в нашей программе.

В чем же проблема? Один из высокоуровневых компонентов имеет программу, которая должна вызываться словом ПЕРЕВОД-СТРАНИЦЫ. А именно, в случае, если таблица или рисунок появляется в середине текста, а время перевода формата еще не подошло, форматтер откладывает рисунок до следующей страницы, продолжая печатать текст. Такое действие требует возможности как-то "забираться" внутрь слова ПЕРЕВОД-СТРАНИЦЫ таким образом, чтобы в следующий раз оно выдавало бы распечатку отложенного рисунка в вершине новой страницы:

: НОВАЯ-СТРАНИЦА ... (закончить страницу с подпечаткой)
(начать новую страницу с надпечаткой) ...?ОТЛОЖЕННОЕ ... ;

Рис.3-10. Два способа наращивания возможностей.



Как может ПЕРЕВОД-СТРАНИЦЫ вызывать ?ОТЛОЖЕННОЕ, если последнее определено гораздо позже?

Хотя теоретически возможно организовать загрузку программы так, чтобы мощные функции вводились до корневых слов, такой подход плох по двум причинам.

Во-первых, разрушается естественная организация (по возрастающей мощности). Во-вторых, мощные функции часто используют код, который определен между элементарными словами. Если вы перемещаете мощные программы к началу, Вам приходится смещать туда же и все используемые ими слова, либо дублировать их код. Чрезвычайно бардачно.

Программирование по принципу уменьшения сложности можно организовать, используя технику "векторизации". Вы можете позволить корневой функции вызывать (указывать на) любую из различных программ, которые должны быть определены после

самой этой функции. В нашем примере заранее необходимо создать только `имя` программы ?ОТЛОЖЕННОЕ, его определение может быть дано позднее.

В главе 7 рассматривается вопрос о векторизации в Форте.

ОГРАНИЧЕННОСТЬ МЫШЛЕНИЯ ПО УРОВНЯМ

Большинство из нас виновны в преувеличении разницы между "высоким уровнем" и "низким уровнем". Такое разделение весьма спорно. Оно ограничивает нашу способность к здравым суждениям по проблемам программирования.

Мышление по уровням, в его традиционном виде, вносит искажения тремя способами:

7. Настаивает на том, чтобы разработка следовала структурной иерархии
8. Настаивает на том, чтобы уровни были отделены друг от друга, исключая этим возможность применения преимуществ повторного использования
9. Поощряет синтаксические различия между уровнями (например, ассемблер против "высокоуровневых" языков) и веру в то, что природа программирования как-то меняется, если уходить все дальше от машинного кода.

Давайте разберем одно за другим каждое из этих заблуждений.

С ЧЕГО НАЧАТЬ?

Я спросил Мура, как бы он подошел к разработке конкретной задачи - игры для детей. Когда дитя нажимает цифры на цифровой клавиатуре, от нуля до девяти, на экране появляется такое же количество больших квадратов.

Мур:

> Я не начинаю с вершины и не прорабатываю задачу вниз. При получении такой ясной задачи я бы начал с написания слова, которое рисует квадрат. Я начал бы с низа и закончил бы словом ИДИ, которое бы обслуживало клавиатуру.

Насколько много в этом интуитивного?

> Быть может, кое-что есть. Я знаю, куда направляюсь, так что мне именно с этого начинать необязательно. И, к тому же, забавнее рисовать квадратики, чем программировать клавиатуру. Я буду делать то, что приятнее всего для того, чтобы углубиться в задачу. Если мне впоследствии придется стереть все эти детали, то это та цена, которую я плачу.

Вы защищаете подход по принципу "наибольшей приятности"?

> Если Вы делаете это в свободной манере, то да. Если бы нам было нужно через два дня показывать это заказчику, я бы делал это по-другому. Я бы начал с самой заметной вещи, а вовсе не с самой забавной. Но все равно не в иерархической последовательности, сверху вниз. Я основываю свой подход на более насущных соображениях типа: произвести впечатление на покупателя, заставить что-либо работать или показать другим людям, как оно будет работать с тем, чтобы их заинтересовать.

Если определить уровень как "вложенность", тогда да, это хороший путь для декомпозиции задачи. Но я никогда не видел пользы от употребления выражения "уровень". Другим аспектом уровней являются языки, мета-языки, мета-мета-языки.
Пытаться разобраться в том, на каком уровне Вы находитесь - на ассемблерном, первом интеграционном или последнем интеграционном уровне - утомительно и мало помогает. Все мои уровни находятся в счастливом смешении.

Проектирование по компонентам делает мало значащим то место, с которого Вы начинаете. Можно было начать с интерпретатора клавиатуры, к примеру. Его целью является получение нажатий на клавиши и преобразование их в числа, с передачей этих чисел вызываемому извне слову. Если Вы замените его словом Форта "." ("точка" распечатывает число со стека), то сможете реализовать интерпретатор клавиатуры, проверить его и отладить без использования программ, имеющих что-либо общее с рисованием квадратов.

С другой стороны, если задача требует поддержки аппаратуры (например, графический пакет), каковой поддержки мы не имеем или не можем купить, то может понадобиться замена ее на что-то доступное, такое, как печать звездочками, для того, чтобы пощупать задачу. Мышление в терминах лексиконов подобно рисованию большой панорамы, состоящей из нескольких полотен. Вы работаете над каждым из полотен по отдельности, вначале набрасывая ключевые элементы сюжета, а затем добавляя цветные мазки здесь и там ... до тех пор, пока вся стена не будет закончена.

СОВЕТ

Решая, с чего начать проектирование, ищите:

- места, для которых требуется максимум творчества (места, где вероятность изменений наиболее велика)
- места, которые дают самую удовлетворительную отдачу (пусть фрукты сочатся)
- места, которые могут в дальнейшем наиболее сильно повлиять на другие области или которые определяют, может ли задача вообще быть разрешена
- вещи, которые следует продемонстрировать заказчику для установления взаимопонимания
- вещи, которые можно показать тем, кто дает деньги, если это нужно для продолжения финансирования.

БЕЗ ПРЕДСТАВЛЕНИЯ НЕТ РАЗДЕЛЕНИЯ.

Второй путь, по которому уровни могут затруднять принятие оптимальных решений - это подталкивание к разделению на уровни. Популярная в проектировании конструкция, называемая "объектом" - типична для этой опасной филисофии.

Объект - это порция кода, которую можно вызвать по одному ее имени, но которая может выполнять более, чем одну функцию. Для выбора определенной функции следует вызвать объект и передать ему параметр или группу параметров. Вы можете представить себе параметры как ряд кнопок, которые можно надавливать для того, чтобы объект делал, что Вы хотите.

Выигрыш от проектирования задачи в терминах объектов состоит в том, что, как и компонент, объект упрятывает информацию от остальной задачи, облегчая обзор.

Несмотря на это, имеются несколько осложнений. Во-первых, объект должен содержать сложную структуру решений для определения, какую из функций ему выполнять. Это увеличивает объем объектного кода и снижает производительность. Лексикон же, со своей стороны, дает Вам все нужные функции при прямом вызове их по именам.

Во-вторых, объект обычно проектируется для автономного выполнения. Он не может использовать преимущества использования инструментария из компонентов поддержки. В результате в нем имеется тенденция к дублированию того кода, который появляется и в других частях задачи. Некоторым объектам даже предлагается разбирать входной текст для интерпретации своих параметров. Каждый из них может иметь свой синтаксис. Позорная трата времени и энергии!

Наконец, поскольку объект конструируется так, чтобы иметь конечный перечень возможностей, трудно делать добавления к его ряду готовому кнопок, когда нужна новая. Инструменты внутри объекта не спроектированы для повторного использования.

Идея уровней пронизывает дизайн моего собственного персонального компьютера, IBM PC. Кроме самого процессора (с его собственным набором машинных команд, разумеется), имеются программные уровни:

- набор утилит, написанных на ассемблере и прожженных в системном ПЗУ
- дисковая операционная система, вызывающая утилиты
- высокоуровневый язык директив, который вызывает операционную систему и утилиты
- и, наконец, любая задача, использующая язык.

Утилиты в ПЗУ предоставляют зависимые от аппаратуры программы: работающие с видеоэкраном, дисководами, клавиатурой. Их вызывают, помещая управляющий код в определенный регистр и генерируя подходящее программное прерывание.

К примеру, программное прерывание 10H вызывает вход в набор программ для работы с изображением. Имеются 16 таких программ. Вы загружаете регистр AH номером желаемой функции.

К сожалению, изо всех 16-ти программ нет ни одной, печатающей строку текста. Для того, чтобы это сделать, Вы должны повторять процесс загрузки регистров и генерации программного прерывания, которое, в свою очередь, должно решать, о какой из программ идет речь и проделывать еще несколько других вещей, которые Вам не требуются - для `каждого отдельного символа`.

Попробуйте написать текстовый редактор, в котором весь экран нужно обновлять при каждом нажатии на клавишу. Работает медленно, как почта! Нельзя улучшить скорость работы, поскольку нельзя повторно использовать никакую информацию внутри видео-программ, кроме той, которая дана для наружного использования. Обоснованной причиной этого является `изоляция` программиста от адресов устройств и других деталей аппаратуры. Ведь все это может измениться при будущих улучшениях.

Единственный способ эффективной реализации ввода/вывода изображения на этой машине - это записывать строки непосредственно в видеопамять. Это можно легко сделать, поскольку руководство по эксплуатации рассказывает об адресах, с которых начинается видеопамять. Однако это разбивает все усилия проектировщиков системы. Ваш код может не пережить замены аппаратуры.

Предполагая `защитить` программиста от деталей, разделение убило цель упрятывания информации. Компоненты же, наоборот, не являются выделенными модулями, но лишь добавлениями, приплюсованными к словарю. Видеолексикон мог бы, в конце концов, давать имя адреса видеопамяти.

Нет ничего неправильного в концепции интерфейсного выбора исполняемых функций между компонентами, если это необходимо.

Проблема здесь состоит в том, что видеокомпонент был спроектирован некомплектно. С другой стороны, если бы система была полностью интегрирована - операционная система и драйверы написаны на Форте - видеокомпонент не `должен` был бы быть спроектирован для ответа на любые потребности. Программист конкретной задачи мог бы либо переписать драйвер, либо написать расширение к драйверу с использованием подходящих инструментов из видеолексикона.

СОВЕТ
Не хороните свои инструменты.

ГОРА ЧЕПУХИ.

Заключительным заблуждением, подготовленным уровневым мышлением, является то, что языки программирования должны меняться качественно при "повышении" их уровня. Мы пытаемся говорить о высокоуровневом коде как о чем-то утонченном, а о низкоуровневом - как о чем-то грубом и простецком.

До некоторой степени такие различия имеют под собой почву, но это лишь результат произвольных архитектурных ограничений, которые все мы принимаем за норму. Мы взращены на ассемблерах, имеющих сжатые мнемоники и ненатуральные синтаксические правила, поскольку они "низкого уровня".

Концепция компонентов восстает против поляризации на высокий и низкий уровень. Весь код должен выглядеть и вести себя одинаково. Компонент - это просто набор команд, которые вместе преобразуют структуры данных и алгоритмы в полезные функции. Эти функции могут быть использованы без знания структур и/или алгоритмов, их составляющих.

Дистанция между этими структурами и настоящим машинным кодом к делу не относится. Код, написанный для манипуляции битами в выходном порту, теоретически, выглядит не более пугающим, нежели код для форматирования докладов.

Даже машинный код должен быть удобочитаем. По-настоящему основанная на Форте машина имела бы синтаксис и словарь, единообразный и идентичный "высокоуровневому" словарю, известному нам сегодня.

РЕЗЮМЕ

В этой главе мы рассмотрели два пути разбиения задачи: на компоненты и в соответствии с возрастающей сложностью.

Особое внимание должно уделяться тем компонентам, которые служат интерфейсами между другими компонентами.

Теперь, если Вы правильно произвели предварительное проектирование, в вашу задачу входит взобраться на кучу управляемых кусочков. Каждый из них представляет задачу, которую надо решить. Выберите Ваш любимый кусок и обращайтесь к следующей главе.

ДЛЯ ДАЛЬНЕЙШЕГО РАЗМЫШЛЕНИЯ

(Ответы приведены в приложении Д)

1. Ниже приведены два подхода к определению интерпретатора клавиатуры. Какой из них предпочли бы Вы? Почему?

А) (Определение клавиш)

```
HEX
72 CONSTANT ВВЕРХКУРС
80 CONSTANT ВНИЗКУРС
77 CONSTANT ВПРАВОКУРС
75 CONSTANT ВЛЕВОКУРС
82 CONSTANT ВСТАВКА
83 CONSTANT ЗАБОЙ

( Интерпретатор клавиш)
: РЕДАКТОР
BEGIN  ЕЩЕ   WHILE   KEY   CASE
    ВВЕРХКУРС  OF   КУРС-ВВЕРХ    ENDOF
    ВНИЗКУРС   OF   КУРС-ВНИЗ    ENDOF
    ВПРАВОКУРС OF   КУРС-ВПРАВО   ENDOF
    ВЛЕВОКУРС  OF   КУРС-ВЛЕВО    ENDOF
    ВСТАВКА    OF   УСТ-ВСТАВКУ   ENDOF
    ЗАБОЙ      OF  СТИРАНИЕ     ENDOF
ENDCASE   REPEAT ;
```

Б) (Интерпретатор клавиш)

```
: РЕДАКТОР
BEGIN  ЕЩЕ   WHILE   KEY   CASE
    72   OF   КУРС-ВВЕРХ    ENDOF
    80   OF   КУРС-ВНИЗ     ENDOF
    77   OF   КУРС-ВПРАВО   ENDOF
    75   OF   КУРС-ВЛЕВО    ENDOF
    82   OF   УСТ-ВСТАВКУ   ENDOF
    83   OF   СТИРАНИЕ      ENDOF
ENDCASE   REPEAT ;
```

2. Эта задача - упражнение по упрятыванию информации.

Предположим, имеется район памяти вне словаря Форта, которые мы хотим зарезервировать под структуры данных (по какой-либо причине). Участок начинается с шестнадцатеричного адреса C000.
Мы хотим определить последовательности массивов, которые будут находиться в данной памяти.

Мы могли бы сделать что-то вроде:

```
HEX
C000 CONSTANT ПЕРВЫЙ-МАССИВ  ( 8 байтов)
C008 CONSTANT ВТОРОЙ-МАССИВ  ( 6 байтов)
```

```
C00C CONSTANT ТРЕТИЙ-МАССИВ  ( 100 байтов)
```

Определенные выше имена массивов будут возвращать начальные адреса соответствующих массивов. Однако заметьте, что нам пришлось вычислять правильный начальный адрес для каждого из них, основываясь на знании того, сколько байтов уже зарезервировано. Давайте попытаемся автоматизировать это, введя "указатель резервирования" по имени >ПАМЯТЬ, который указывает на следующий свободный байт. Вначале мы устанавливаем указатель на начало места в памяти:

```
VARIABLE >ПАМЯТЬ
C000 >ПАМЯТЬ !
```

Теперь мы можем определить каждый из массивов так:

```
>ПАМЯТЬ @   CONSTANT   ПЕРВЫЙ-МАССИВ    8 >ПАМЯТЬ +!
>ПАМЯТЬ @   CONSTANT   ВТОРОЙ-МАССИВ    6 >ПАМЯТЬ +!
>ПАМЯТЬ @   CONSTANT   ТРЕТИЙ-МАССИВ  100 >ПАМЯТЬ +!
```

Заметьте, что после определения каждого из массивов мы увеличиваем указатель на размер этого массива, чтобы показать, что мы отвели для него столько дополнительной памяти.

Для большей удобочитаемости вышеприведенного мы должны добавить такие два определения:

```
: ТАМ  ( -- адрес-следующего-свободного-байта-в-ОЗУ)
    >ПАМЯТЬ @ ;
: ДАТЬ-ОЗУ  ( #байтов-для-массива -- )  >ПАМЯТЬ +! ;
```

Мы можем теперь переписать то же самое как:

```
ТАМ  CONSTANT  ПЕРВЫЙ-МАССИВ     8 ДАТЬ-ОЗУ
ТАМ  CONSTANT  ВТОРОЙ-МАССИВ     6 ДАТЬ-ОЗУ
ТАМ  CONSTANT  ТРЕТИЙ-МАССИВ   100 ДАТЬ-ОЗУ
```

(Опытный Форт-программист, скорее всего, скомбинировал бы все эти операции в единое определяющее слово, однако это не то, к чему я подвожу.)

Наконец, предположим, что у нас имеется 20 таких определений массивов, разбросанных по всему тексту.

Теперь задача: вдруг меняется архитектура нашей системы и мы решаем, что должны отвести эту память так, чтобы она `заканчивалась` на шестнадцатеричном адресе EFFF. Другими словами, мы должны начинать с конца, отводя массивы в обратном порядке. Мы при этом все равно хотим, чтобы имя массива возвращало его `начальный` адрес.

Чтобы проделать это, нам теперь нужно написать:

```
F000 >ПАМЯТЬ !  ( последний байт EFFF плюс 1)
: ТАМ  ( -- адрес-следующего-свободного-байта-в-ОЗУ)
  >ПАМЯТЬ @ ;
: ДАТЬ-ОЗУ  ( #байтов-под-массив -- )  NEGATE >ПАМЯТЬ +! ;
   8 ДАТЬ-ОЗУ  ТАМ CONSTANT   ПЕРВЫЙ-МАССИВ
  6 ДАТЬ-ОЗУ  ТАМ CONSTANT   ВТОРОЙ-МАССИВ
  100 ДАТЬ-ОЗУ  ТАМ CONSTANT   ТРЕТИЙ-МАССИВ
```

На этот раз ДАТЬ-ОЗУ `уменьшает` указатель. Все нормально, легко добавить NEGATE в определение ДАТЬ-ОЗУ. Беспокойство вызывает только то, что мы должны ДАТЬ-ОЗУ `до` определения массива, а не после. В нашей программе необходимо найти и исправить двадцать мест.

Слова ТАМ и ДАТЬ-ОЗУ хороши и приятны, но не скрывают информации о том, `как` отводится место. Если бы они это делали, не имело бы значения, в каком порядке их вызывают.

И вот, наконец, наш вопрос: что мы могли бы сделать со словами ТАМ и ДАТЬ-ОЗУ для минимизации влияния изменений в проекте? (Опять же, ожидаемый мною ответ не должен опираться на определяющие слова.)

ГЛАВА 4

ДЕТАЛИЗИРОВАННАЯ РАЗРАБОТКА

РЕШЕНИЕ ЗАДАЧИ

`Тривиально`: Я вижу, как это можно сделать. Я только не знаю, сколько это займет времени.
`Нетривиально`: У меня нет `ключа к пониманию` того, как это сделать.

`Операционная философия, разработанная в группе лабораторной автоматизации и инструментального проектирования на химическом факультете Политехнического института и Университета штата Вирджиния.`

Когда Вы приняли решение по составу компонентов в задаче, Вашим следующим шагом является разработка этих компонентов. В этой главе мы применим технику решения задач для детализированной проработки проблемы на Форте. Это - время чистого творчества, та часть, которую большинство из нас находят наиболее занятной.

Есть особое моральное удовлетворение в том, чтобы выйти на ринг с нетривиальной проблемой и уйти оттуда победителем.

Трудно на естественном языке отделить идею от слов, используемых для ее описания. При написании задачи на Форте трудно отделить детализированную разработку от реализации, поскольку мы стараемся проектировать на Форте. По этой причине мы немного обгоним самих себя в этой главе, не только представляя проблему, но также проектируя решение для нее, прямо ведущее к кодированию реализации.

ТЕХНИКА РЕШЕНИЯ ЗАДАЧ

Даже неофиты могут решать программные задачи без малейших размышлений по поводу проблемы решения задач. Так почему же надо изучать такую технику? Для ускорения процесса. Думая о `путях`, по которым мы решаем проблемы, отдельно от собственно проблем, мы обогащаем наш подсознательный технологический багаж.

Г. Полиа написал насколько книг на тему решения задач, в особенности математических. Наиболее подходящей из них является `Как ее решить` [1]. Хотя решение математических задач не совсем то же самое, что и решение программных, Вы найдете здесь некоторые ценные предложения.

Следующая серия советов суммирует несколько приемов, рекомендуемых наукой о решении задач:

СОВЕТ
Определите Вашу цель.

Знайте, что Вы пытаетесь сделать. Как мы видели в главе 2, этот шаг может быть детализирован и дальше:

Определите интерфейсы данных: проследите, какие данные понадобятся для достижения цели и убедитесь, что эти данные доступны (вход). Для одного определения это означает написание стекового комментария.

Определите правила: проверьте все известные Вам факты. В главе 2 мы описывали тарифы для вычисления стоимости телефонного вызова с помощью правил для применения тарифов.

СОВЕТ

Обрисуйте проблему в целом.

В фазе `анализа` мы разделили задачу на части для выяснения нашего понимания каждой из этих частей. Теперь мы входим в фазу `синтеза`. Мы должны визуализировать проблему в целом.

Попытайтесь удержать в голове столько информации о проблеме, сколько это возможно. Используйте слова, фразы, рисунки и таблицы или любой вид графического представления данных и/или правил для того, чтобы помочь себе одним взглядом охватывать как можно больше информации. Ощущайте, как распухает Ваша голова от требований, которые вы должны выполнить в задаче, подобно тому, как ощущаете наполнение своих легких воздухом.

Теперь задержите Ваш мысленный образ, так же, как вы задерживаете дыхание.

Случится одно из двух:

Вы можете увидеть решение во вспышке озарения. Прекрасно! Испустите вздох облегчения и приступайте прямо к реализации.

Или ... задача слишком сложна или незнакома, чтобы ее можно было решить так легко. В этом случае Вам следует обратить свое внимание на аналоги и частичные решения. При этом важно то, что вы уже сконцентрировались на требованиях задачи в целом.

СОВЕТ

Разработайте план.

Если решение не очевидно, следующим шагом является определение подхода, который Вам следует принять для его выработки.

Установите направление действий и остерегайтесь ловушек бесцельного шатания.

Следующие советы предлагают несколько подходов, которые Вам стоило бы принять.

СОВЕТ

Подумайте об аналогичной задаче.

Нет ли чего-то знакомого в этой проблеме? Не писали ли Вы подобного определения раньше? Выделите знакомые участки проблемы и то, в чем может

отличаться данная задача. Попытайтесь вспомнить, как Вы решили ее раньше или как Вы решили раньше что-то похожее.

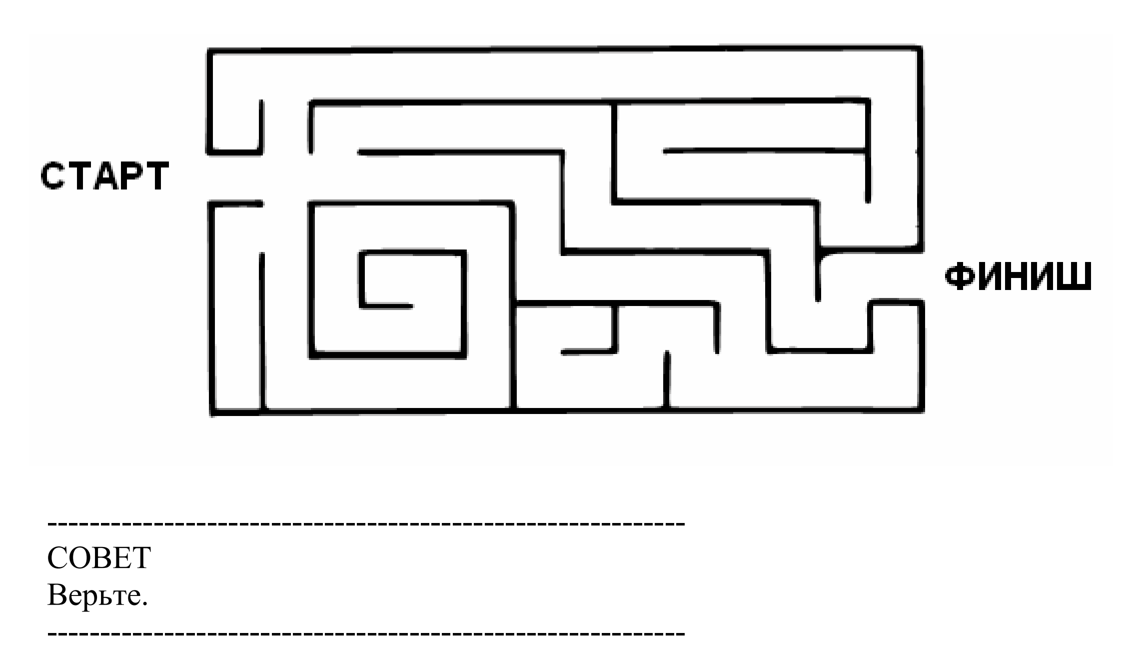
СОВЕТ
Работайте назад.

Нормальный, очевидный путь для атаки на проблему начинается с области известного и продолжается в неизвестном. В решении о том, на какую лошадь ставить, Вы бы начали с их недавнего прошлого, с их текущего здоровья и так далее, применяя разные веса для этих различных факторов и приближаясь к фавориту.

СОВЕТ
Работайте назад.

Более сложные проблемы предоставляют множество возможных путей для обращения с поступающими данными. Как узнать, какой маршрут приведет Вас ближе к решению? Неизвестно. Этот класс задач лучше решается при проработке назад (рисунок 4-1).

Рис.4-1. Задача, которую проще решать назад, чем вперед.

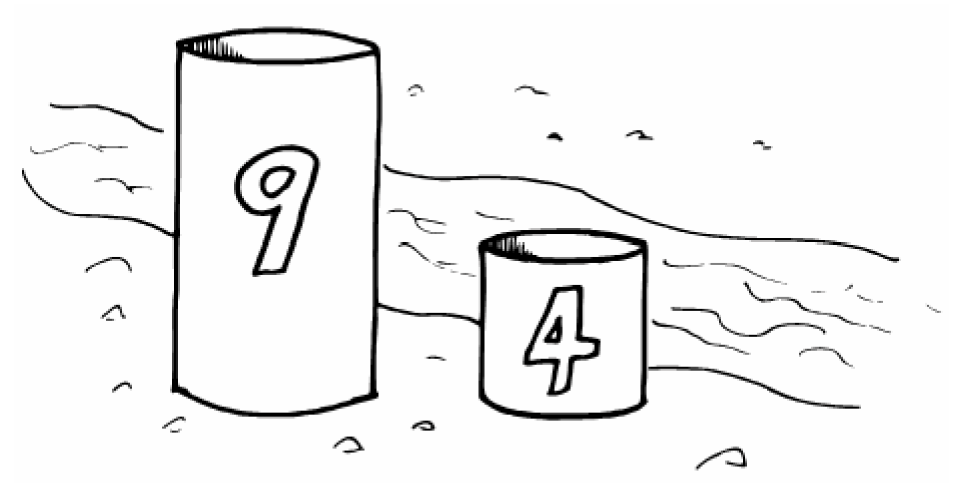


СОВЕТ
Верьте.

Вера является обязательной составляющей для успешной проработки назад. Мы проиллюстрируем это с помощью знаменитой математической задачи. Предположим, у нас есть две бочки.

На них нет градуировочных отметок, однако одна имеет емкость в девять ведер, а другая - четыре ведра. Нашей задачей является отмеривание ровно шести ведер воды из рядом текущего источника в одну из бочек (рисунок 4-2).

Рис.4-2. Две бочки.



Попытайтесь решить задачу сами, прежде чем читать дальше.

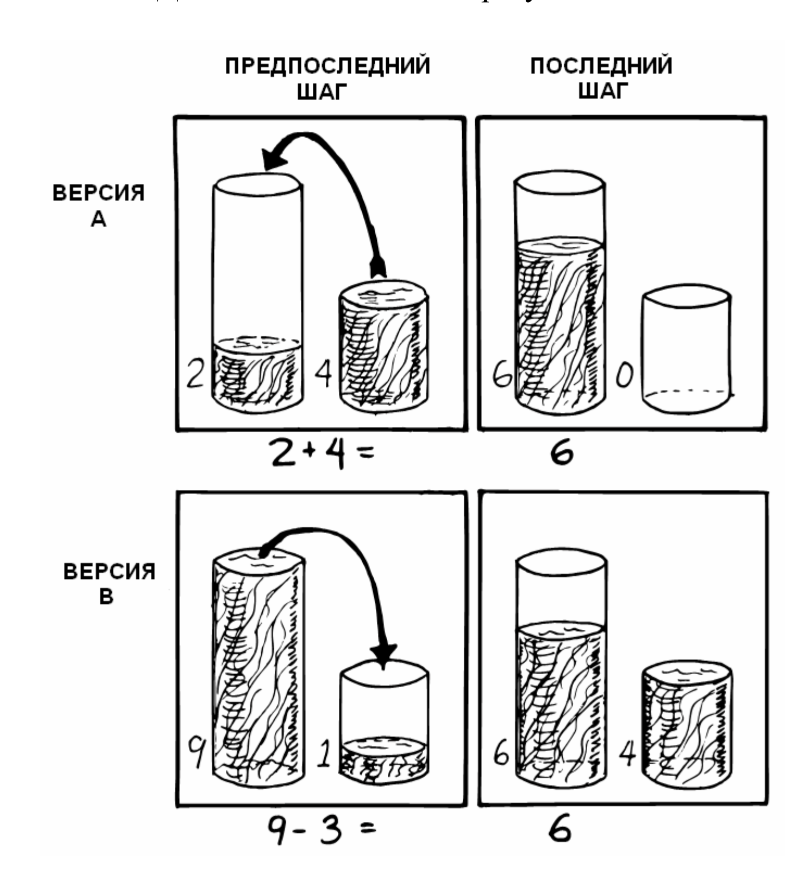
Как мы можем получить "шесть" из "девяти" и "четырех"? Мы можем начать проработку вперед, переливая в уме воду из одной бочки в другую. К примеру, если мы наполним большую бочку дважды из маленькой, то получим восемь ведер. Если мы наполним доверху девятиведерную, затем перельем из нее достаточно воды, чтобы заполнить четырехведерную, у нас будет ровно пять ведер в большой бочке.

Это интересные идеи, однако они не принесли нам шести ведер. И не ясно, как они могут это сделать.

Давайте попытаемся пойти назад. Договоримся, что мы намеряли шесть ведер воды, и она содержится в большей бочке (она не влезет в меньшую!). Теперь, как мы этого достигли? Каково было состояние наших бочек один шаг назад?

Имеются лишь две возможности (рисунок 4-3):

Рис.4-3. Достижение конечного результата.



1. Четырехведерная была полна, и мы лишь добавили ее в большую. Это предполагает, что мы уже имели два ведра в большой бочке. Или ...
2. Девятиведерная бочка была полна, и мы попросту перелили из нее три ведра в маленькую.

Что выбрать? Давайте поразмыслим. Выбор первого варианта требует двухведерного измерения, второй требует трехведерного.

При нашем начальном проигрывании ситуации мы не встречались с такой единицей, как два. Однако мы видели разницу в один, и один из четырех есть три. Давайте остановимся на версии Б.

Теперь начинается настоящий фокус. Мы должны заставить себя `поверить` без малейшего сомнения в то, что мы достигли описанной ситуации. Мы перелили три ведра в малую бочку.

Отбрасывая все свое неверие, мы концентрируемся на том, как мы это сделали.

Как мы смогли перелить три ведра в бочонок? Если там уже было одно такое ведро! Неожиданно мы перевалили через гору.

Теперь остался простой вопрос о том, как мы получили одно ведро в маленькой бочке? Мы должны были начать с полной девятиведерной, отлить дважды по четыре ведра и получить одно.

Затем мы перелили это ведро в маленькую бочку.

Наш последний шаг должен будет заключаться в проверке нашей логики проигрыванием задачи вперед опять.

Вот еще одно преимущество проработки назад: если задача нерешаема, проработка назад помогает Вам быстро доказать отсутствие ее решения.

СОВЕТ

Обнаруживайте внешние проблемы.

До тех пор, пока мы не порешали проблему, у нас имелось лишь смутное представление о том, какие шаги - или даже сколько шагов - может понадобиться. По мере того, как мы больше проникаемся задачей, мы начинаем понимать, что она содержит одну или более подзадач, которые выглядят несколько отличными от главной линии преполагаемых действий.

В только что решенной задаче мы обнаружили две подзадачи: заливку бочонка одним ведром воды и затем наполнение большой бочки шестью ведрами.



Intent on a complicated problem.

Распознавание таких маленьких проблем, иногда называемых "внешними задачами", является важной частью техники решения задач. Определяя подзадачу, мы можем допускать, что она имеет прямое решение. Не останавливаясь для поиска этого решения мы устремляемся вперед по нашей главной задаче.

(Форт, как мы увидим, идеально приспособлен для применения такой техники.)

СОВЕТ

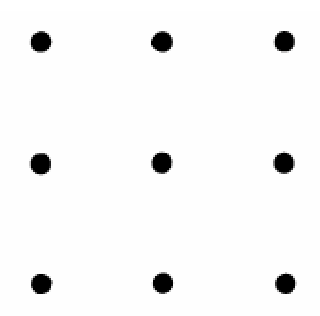
Отступите от проблемы.

Так легко поддаться чувствам и привязаться к одному конкретному решению, что мы забываем держать ум открытым.

Литература по решению задач часто использует пример девяти точек. В свое время она поставила меня в тупик, так что я приведу ее. Имеются девять точек, расставленных, как показано на рисунке 4-4. Надо нарисовать прямые линии так, чтобы они прошли через все девять точек, не отрывая при этом перо от бумаги.

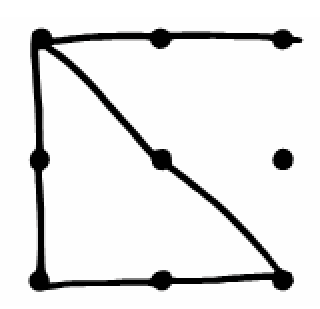
Ограничение состоит в том, что сделать это нужно всего четырьмя линиями.

Рис.4-4. Задача о девяти точках.



Вы можете хорошенько посидеть и получить что-то не лучшее, нежели почти правильный рисунок 4-5. Если Вы действительно сильно сконцентрируетесь, то можете заключить, что задача шуточная - у нее нет решения.

Рис.4-5. Не совсем правильно.



Но если Вы отсядете от нее и спросите себя,

"Не обманываю ли я себя удобным маневром из-за узости своего мышления? Не учитываю ли я некоторые ограничения, которых нет в описании проблемы? Каковы могут быть эти ограничения?" тогда Вы сможете догадаться вывести некоторые линии за периметр квадрата девяти точек.

СОВЕТ
Используйте мышление на полную катушку.

Когда проблема загнала Вас в тупик и Вы оказываетесь ни с чем, расслабьтесь, перестаньте беспокоиться об этом, быть может, даже забудьте об этом на время.

Творческие люди всегда отмечали, что лучшие идеи приходят к ним в постели или под душем. Множество книг по решению задач предлагают полагаться на подсознание для разрешения действительно сложных проблем.

Современные теории о функциях мозга исследуют различия между рациональным, сознательным мышлением (которое опирается на манипуляции с символами) и подсознательным мышлением (которое проводит корелляцию между новыми познаниями и ранее усвоенной информацией, перекомбинируя и переделывая связи новым и полезным образом).

Лесли Харт [2] объясняет трудность решения больших задач с помощью логики:

Огромная нагрузка ложится на ту маленькую функцию мозга, на которой может быть сосредоточено внимание в данный момент.

Возможно озарение, подобное цирковому фокусу, и больше похоже на то, что при этом ... используются все ресурсы нашей великолепной подкорки, ... многомиллиардная нейронная емкость мозга.

... Рабочая часть лежит в снабжении мозга рядами входных данных, таких, как наблюдение, чтение, сбор сведений и просмотр того, что достигли другие. Когда все это введено, приходит время [подсознательных] процедур, суммарных, автоматических, лежащих вне зоны внимания.

... Кажется очевидным, ... что поиск производится за некоторый интервал времени, хотя и необязательно непрерывный, что во многом напоминает работу большого компьютера. Я бы рискнул предположить, что поиск ветвится, начинается, оканчивается, достигает тупиков и возвращается назад и в конце концов собирает ответ, который оформляется и затем всплывает в сознательном внимании - зачастую в удивительно проработанном виде.

СОВЕТ
Сформируйте Ваше решение. Поищите другие решения.

Вы смогли найти один способ. Могут быть и другие, и некоторые из них могут быть лучше.

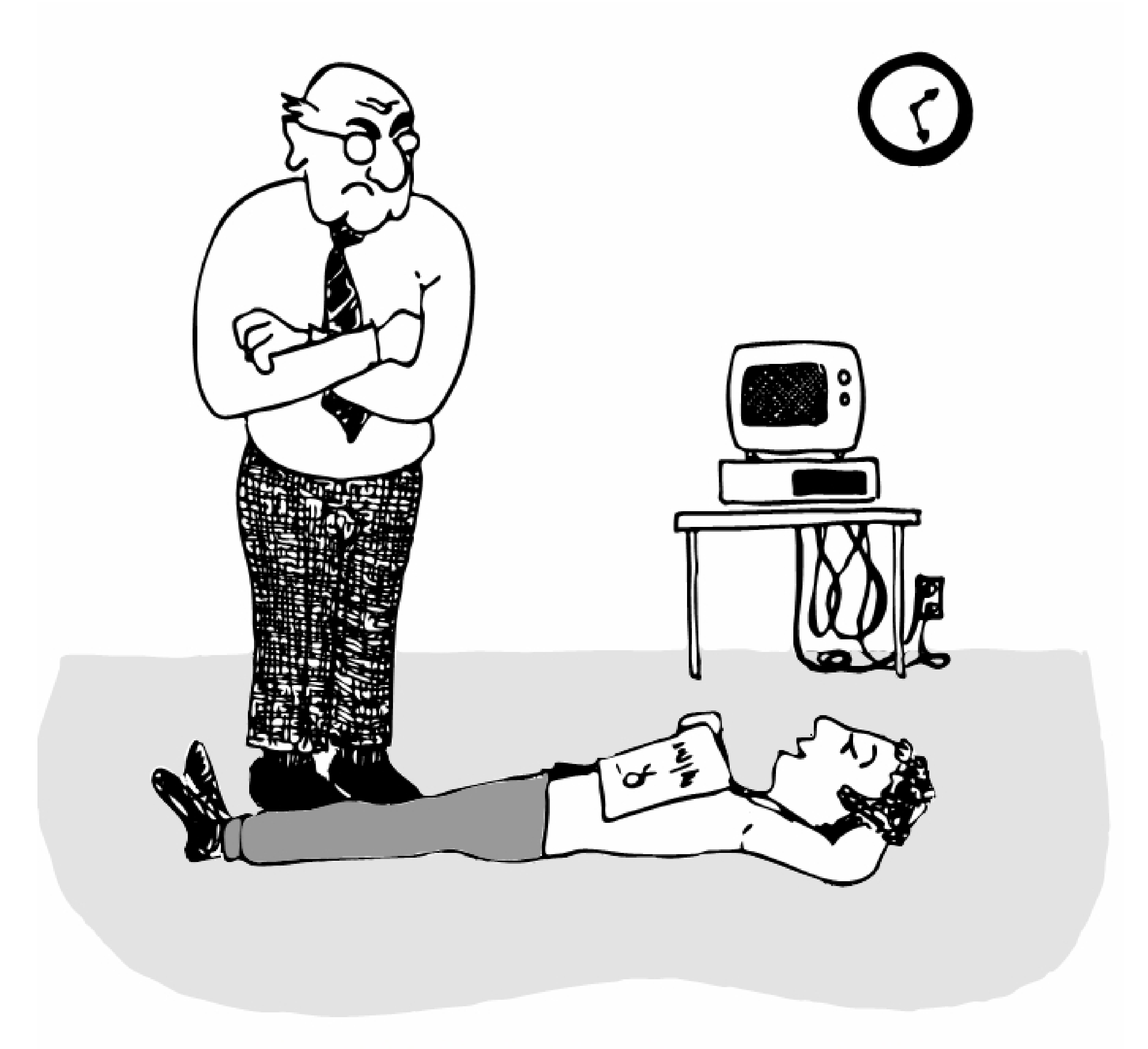
Не вкладывайте слишком много усилий в свое первое решение, пока не попытаетесь выработать в себе иное мнение.

ИНТЕРВЬЮ С ИЗОБРЕТАТЕЛЕМ-ПРОГРАММИСТОМ

Дональд А. Барджисс, владелец и президент фирмы Scientek Instrumentation, Inc.:

У меня есть несколько приемов, которые я с пользой применял многие годы при проектировании чего бы то ни было, и которые позволяют мне сохранять гибкость. Моим

первым правилом является: "Ничто не невозможно". Мое второе правило - "Не забывать, что главное - добиться желаемого".



"I'm not just sleeping. I'm using my neocortex."

Вначале исследуйте проблему, набрасывая на бумаге два или три подхода к ней. Затем попробуйте кажущийся наиболее подходящим из них, чтобы посмотреть, работает ли он. Проведите его. Затем вдумчиво проделайте весь путь назад и начните заново.

Начинать заново хорошо по двум причинам. Во-первых, это дает вам свежий подход. Вы можете быть притянуты назад тем путем, с которого начинали, либо Ваше начинание притянется к новому пути.

Во-вторых, новый подход может продемонстрировать все виды мощных возможностей. Вы имеете теперь средство измерения. Можете взглянуть на оба подхода и сравнить преимущества обоих. Вы оказываетесь в лучшей позиции для суждения.

Попадание в тупик происходит от слишком уж настойчивых попыток следовать единственному подходу. Напоминайте себе: "Я хочу изменить эту зубодробильную штуку. Отбросим традиционный подход как не представляющий интереса.

Попробуем безумные идеи." Лучше всего начать рисовать картинки. Я рисую мелких человечков. Это позволяет вещам не выглядеть как "данные" и не идет вразрез с моим мыслительным процессом. Человеческий мозг исключительно хорошо работает на аналогиях.

Представление вещей в контексте удерживает Вас от замыкания в пределах условностей любого языка, даже Форта. Когда я хочу сконцентрироваться на чем-то небольшом, я разрисовываю маленькие кусочки бумаги. Когда я хочу думать широкими мазками, чтобы понять общее направление, я разрисовываю большие куски бумаги. Это все некоторые дурацкие штучки, которые я использую для того, чтобы не застаиваться.

Когда я программирую на Форте, то провожу день только в грезах, бродя вокруг идей. Обычно перед тем, как сесть за клавиатуру, я набрасываю их в общих выражениях. Не код, только слова. Заметки для себя.

Затем я начинаю с последней строки программы. Я описываю, что мне хотелось бы сделать, на языке, как можно более близком к английскому. Затем я использую редактор для смещения этого определения к концу экрана и начинаю кодирование внутренних слов. Далее я осознаю, что такой путь не подходит. Может быть, я разобью свое основное слово на два и перемещу одно из них в более ранний блок так, что смогу его использовать раньше. Я работаю с аппаратурой, если она есть в наличии, иначе я ее симулирую.

Форт требует самодисциплины. Вы должны перестать тыкаться в клавиатуру. Форт хочет делать то, что я ему приказываю, а я приказываю ему делать различные виды нелепых вещей, не имеющих ничего общего с тем, куда мне нужно идти. В такие моменты мне нужно убираться от клавиатуры.

Форт позволяет Вам играть. Это здорово, у Вас есть шансы на то, чтобы получить кое-какие идеи. До тех пор, пока Вы удерживаетесь от игры в качестве привычки. Ваша голова гораздо лучше компьютера приспособлена для изобретений.

ДЕТАЛИЗИРОВАННАЯ РАЗРАБОТКА

Теперь мы находимся в той точке цикла разработки, когда уже принято решение о том, что нам нужен компонент (или конкретное слово). Компонент будет состоять из нескольких слов, некоторые из которых (те, что составляют лексикон) будут использоваться другими компонентами, другие же (внутренние слова) будут использованы лишь внутри этого компонента.

Создавайте столько новых слов, сколько нужно для выполнения следующего совета:

СОВЕТ
Каждое определение должно выполнять простую, хорошо определенную задачу.

Вот шаги, обычно предпринимаемые при разработке компонента:

1. Основываясь на требуемых функциях, примите решения об именах и синтаксисе для внешних определений (определите интерфейсы).
2. Разверните концептуальную модель, описав алгоритм(ы) и структуры данных.
3. Очертите круг внешних используемых определений.
4. Установите, какие из внешних определений и приемов уже доступны.
5. Опишите алгоритм с помощью псевдокода,
6. Разработайте его, двигаясь назад от существующих определений к входным данным,

7. Реализуйте все недостающие внешние определения.
8. Если лексикон содержит много имен, имеющих массу общих элементов, разработайте и закодируйте эти общие части как внутренние определения, а затем реализуйте внешние имена.

Мы обсудим глубже два первых шага, затем же рассмотрим пример полной разработки лексикона.

СИНТАКСИС ФОРТА

В этой точке цикла разработки Вы должны решить, как слова в Вашем новом лексиконе будут использоваться в контексте. При этом надо держать в уме то, как лексикон будет использоваться последующими компонентами.

\---

СОВЕТ

При разработке компонента целью является создание лексикона, который сделает Ваш последующий код читаемым и легко управляемым.

\---

Каждый компонент должен разрабатываться с мыслью об использующих его компонентах. Вам надо проектировать синтаксис лексикона так, чтобы слова имели смысл при их появлении в контексте. Упрятывание внутренней информации, как мы уже видели, обеспечит при этом управляемость.

В то же время имейте в виду собственный синтаксис Форта. Вместо того, чтобы внедрять определенный синтакис только потому, что он кажется подходящим, Вы можете уберечь себя от написания большого количества ненужного кода, только выбирая синтаксис, который Форт может поддерживать без какого-либо специального напряжения с Вашей стороны.

Вот некоторые элементарные правила естественного синтаксиса Форта:

\---

СОВЕТ

Пусть числа предшествуют словам.

\---

Слова, требующие числового аргумента, должны, естественно, находить его на стеке. С точки зрения синтаксиса это значит, что число должно предшествовать имени. К примеру, синтаксис слова SPACES, которое выдает "N-ное" количество пробелов, есть:

```
20 SPACES
```

Иногда такое правило нарушает тот порядок, к восприятию которого привыкло наше ухо. К примеру, слово Форта + должно быть предварено двумя аргументами, как в выражении

```
3 4 +
```

Такой порядок, при котором числа предшествуют операторам, называется "постфиксной записью".

Форт, по своему великодушию, не `настаивает` на такой постфиксной нотации. Вы можете переопределить + и получать одно из чисел из входного потока:

3 + 4

написав:

```
: +   BL WORD  NUMBER  DROP   + ;
```

(где WORD - слово из стандарта Форт-79/83, возвращающее адрес; NUMBER же дает число двойной длины - как это указано в списке слов нерегламентированного употребления стандарта-83).



Отлично. Однако Вы не можете использовать это определение внутри других определений через двоеточие или передавать ему аргументы, теряя таким образом одно из основных преимуществ Форта.

Зачастую слова вида "существительное" передают свои адреса (или другой тип указателя) через стек словам типа "глагол".

Фортоподобный синтаксис фразы "существительное" "глагол" в общем случае реализовать легче всего вследствие использования стека.

В некоторых ситуациях такой порядок слов звучит неестественно. К примеру, предположим, у нас есть файл по имени ИНВЕНТАРЬ. Мы, в частности, можем ПОКАЗАТЬ его, форматируя информацию симпатичными колонками. Если ИНВЕНТАРЬ передает указатель слову ПОКАЗАТЬ, которое с ним работает, синтаксис становится таким:

ИНВЕНТАРЬ ПОКАЗАТЬ

Если Ваше задание предусматривает английский (русский) порядок слов, в Форте имеются способы достижения и его. Однако при этом обычно увеличивается уровень сложности. Иногда лучшее, что можно сделать - это использовать более подходящие имена. Как насчет

ИНВЕНТАРНЫЙ СПИСОК

(Мы сделали "указатель" прилагательным, а "исполнителя" - глаголом.) Если же задание настаивает на синтаксисе

ПОКАЗАТЬ ИНВЕНТАРЬ

то мы имеем несколько возможностей. ПОКАЗАТЬ могло бы устанавливать флаг и ИНВЕНТАРЬ при этом мог бы работать в соответствии с этим флагом. У такого

подхода есть определенные недостатки, в особенности тот, что ИНВЕНТАРЬ должен быть достаточно "умен" для того, чтобы знать все возможные действия, которые могут быть с ним произведены. (Мы будем рассматривать эти проблемы в главах 7 и 8.)

Или ПОКАЗАТЬ могло бы выбирать следующее слово из входного потока. Мы обсудим этот вопрос в совете "избегайте упреждающих выборок" позже в этой главе.

Или, что рекомендуется, ПОКАЗАТЬ может устанавливать "исполняемую переменную", которую ИНВЕНТАРЬ затем будет вызывать. (Мы обсудим векторизованное исполнение в седьмой главе.)

СОВЕТ

Пусть текст идет после имен.

Если Форт-интерпретатор обнаруживает строку текста, не являющуюся ни числом, ни предварительно определенным словом, это вызовет аварийный останов с выдачей сообщения об ошибке. По этой причине неопределенная строка должна быть предваряема заранее определенным словом.

Примером является ." (точка-кавычка), предваряющая текст, который должен быть впоследствии напечатан. Другой пример - CREATE (так же, как и все определяющие слова), предваряющее имя, которое на данный момент еще не определено.

Это правило также применимо к определенным словам, на которые Вам нужно сослаться, но не исполнить их. Пример – слово FORGET:

```
FORGET TASK
```

Синтаксически FORGET должно стоять перед TASK, так что TASK не исполняется.

СОВЕТ

Пусть определения поглощают свои аргументы.

Это синтаксическое правило больше относится к соглашению о хорошем стиле программирования на Форте, чем к требованиям самого Форта.

Предположим, Вы пишете слово ЗАПУСТИТЬ, которое требует номер пусковой установки и стреляет нужной ракетой. В целом Вы хотели бы, чтобы оно выглядело как-то вроде:

```
: ЗАПУСТИТЬ ( ракета#)   ЗАРЯДИТЬ ЦЕЛИТЬ СТРЕЛЯТЬ ;
```

Каждое из трех внутренних определений потребует один и тот же аргумент - номер установки. Вам где-то понадобится поставить два слова DUP. Вопрос только: где? Если Вы введете их внутрь ЗАРЯДИТЬ и ЦЕЛИТЬ, то сможете не употреблять их внутри ЗАПУСТИТЬ, как в вышеприведенном определении. Если Вы их извлечете из ЗАРЯДИТЬ и ЦЕЛИТЬ, Вам нужно будет определить:

```
: ЗАПУСТИТЬ ( ракета#) DUP ЗАРЯДИТЬ  DUP ЦЕЛИТЬ  СТРЕЛЯТЬ ;
```

В соответствии с соглашением, последняя версия предпочтительней, поскольку ЗАРЯДИТЬ и ЦЕЛИТЬ получаются чище.

Если бы Вам понадобилось написать слово ГОТОВ, Вы могли бы это сделать так:

```
: ГОТОВ ( ракета#)  DUP ЗАРЯДИТЬ  ЦЕЛИТЬ ;
```

а не

```
: ГОТОВ ( ракета#)  ЗАРЯДИТЬ  ЦЕЛИТЬ   DROP ;
```

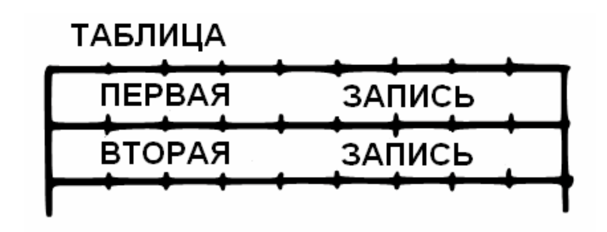
СОВЕТ
Используйте ноль в качестве точки начала отсчета.

По привычке люди нумеруют вещи, начиная с первой: "первая, вторая, третья," и т.д. С другой стороны, математические модели более естественно работают при начале отсчета от нуля.

Поскольку компьютеры являются процессорами чисел, программное обеспечение становится легче писать при использовании нуля в качестве точки отсчета.

Для иллюстрации предположим, что у нас есть таблица 8-байтовых записей. Первая запись занимает первые восемь байтов таблицы. Для вычисления ее начального адреса мы добавляем "0" к адресу ТАБЛИЦА. Для вычисление начального адреса "второй" записи мы добавляем "8" к адресу ТАБЛИЦА.

Рис.4-6. Таблица 8-байтовых записей.



Легко написать формулу для получения таких результатов:

первая запись начинается на: 0 X 8 = 0
вторая запись начинается на: 1 X 8 = 8
третья запись начинается на: 2 X 8 = 16

Мы можем легко написать слово, которое преобразует номер записи в ее стартовый адрес:

```
: ЗАПИСЬ  ( #записи -- адрес)
8 *  ТАБЛИЦА + ;
```

Так в терминах компьютера имеет смысл называть "первой записью" 0-вую запись.

Если постановка Вашей задачи предполагает начало отсчета с единицы, то все нормально. Используйте счет относительно нуля по всей Вашей задаче и затем, только в "лексиконе пользователя" (наборе слов, которые будет употреблять конечный пользователь) сделайте преобразование из одной привязки в другую:

```
: ТЕМА  ( n -- адрес)   1- ЗАПИСЬ ;
```

СОВЕТ
Пусть адреса предшествуют отсчетам.

Опять-таки, это соглашение, а не требование Форта, которое способствуют читаемости кода. Вы найдете примеры применения этого правила в словах TYPE, ERASE и BLANK.

СОВЕТ
Пусть источник стоит перед приемником.

Еще одно соглашение для удобочитаемости. К примеру, в некоторых системах фраза

```
22 37 COPY
```

копирует блок 22 в блок 37. Синтаксис слова CMOVE включает в себя как это, так и предыдущее соглашение:

```
источник  приемник  количество  CMOVE
```

СОВЕТ
Остерегайтесь упреждающих выборок (из входного потока).

В общем случае старайтесь не создавать определений, подразумевающих, что во входном потоке для них будут подготовлены другие слова.

Предположим, в Вашем компьютере с цветным дисплеем синий представлен числом 1, а светло-синий - числом 9. Вы хотите определить два слова: СИНИЙ будет возвращать 1; СВЕТЛО может предварять СИНИЙ для преобразования его в 9.

На Форте можно определить СИНИЙ как константу, так что при исполнении оно будет всегда возвращать требуемую 1.

```
1 CONSTANT СИНИЙ
```

А теперь определим СВЕТЛО так, чтобы оно искало следующее слово во входном потоке, исполняло его и делало логическое "ИЛИ" с восьмеркой (логика этого процесса станет ясной, когда мы вернемся к этому примеру позже в нашей книге):

```
: СВЕТЛО  ( предшествует цвету)  ( -- число цвета)
' EXECUTE   8 OR ;
```

(для FIG-Форта: : СВЕТЛО [COMPILE] ' CFA EXECUTE 8 OR ;)

(Для новичков: апостроф в определении СВЕТЛО - это слово Форта (по-другому - "штрих"). Штрих используется для поиска по словарю; он выбирает имя и ищет его в

словаре, возвращая адрес начала определения. При использовании внутри определения он будет искать слово, следующее за СВЕТЛО - например, СИНИЙ – и передавать этот адрес слову EXECUTE, которое исполняет СИНИЙ, при этом на стеке остается число 1. "Отработав" оператор СИНИЙ, СВЕТЛО теперь делает "ИЛИ" с числом 8, получая 9.)

Это определение будет работать при вызове из входного потока, но понадобятся специальные меры, если мы захотим использовать СВЕТЛО внутри другого определения, как например:

```
: РЕДАКТИРОВАНИЕ      СВЕТЛО СИНИЙ   БОРДЮР ;
```

Даже при работе со входным потоком использование EXECUTE приведет к катастрофе, если после СВЕТЛО случайно будет указано что-нибудь, отличное от нужного определенного слова.

Предпочтительный подход заключается в том, что, если мы вынуждены использовать именно такой синтаксис, то СВЕТЛО должно устанавливать флаг для СИНИЙ, а СИНИЙ должен определять наличие такого флага; это мы позже еще будем рассматривать.

Случается, что заглядывание вперед по входному потоку желательно, даже обязательно. (Предлагаемое далее слово TO часто реализуется именно так [3].)

Но, в общем случае, опасайтесь упреждающих заглядываний. Вам иначе придется готовиться к разочарованиям.

СОВЕТ
Пусть команды исполняют сами себя.

Это правило соотносится с предыдущим. Позволять словам самим делать свою работу - это один из философских кульбитов Форта. Свидетельство тому - компилятор Форта (функция, компилирующая определения через двоеточие), карикатура на который представлена на рисунке 4-7. В нем всего несколько правил:

Брать следующее слово во входном потоке и искать его в словаре.

Если это обычное слово, `компилировать` его адрес.

Если это слово "немедленного исполнения", `исполнить` его.

Если это слово не определено, попытаться преобразовать его в цифру и скомпилировать как литерал.

Если это не число, выполнить аварийный выход с сообщением об ошибке.

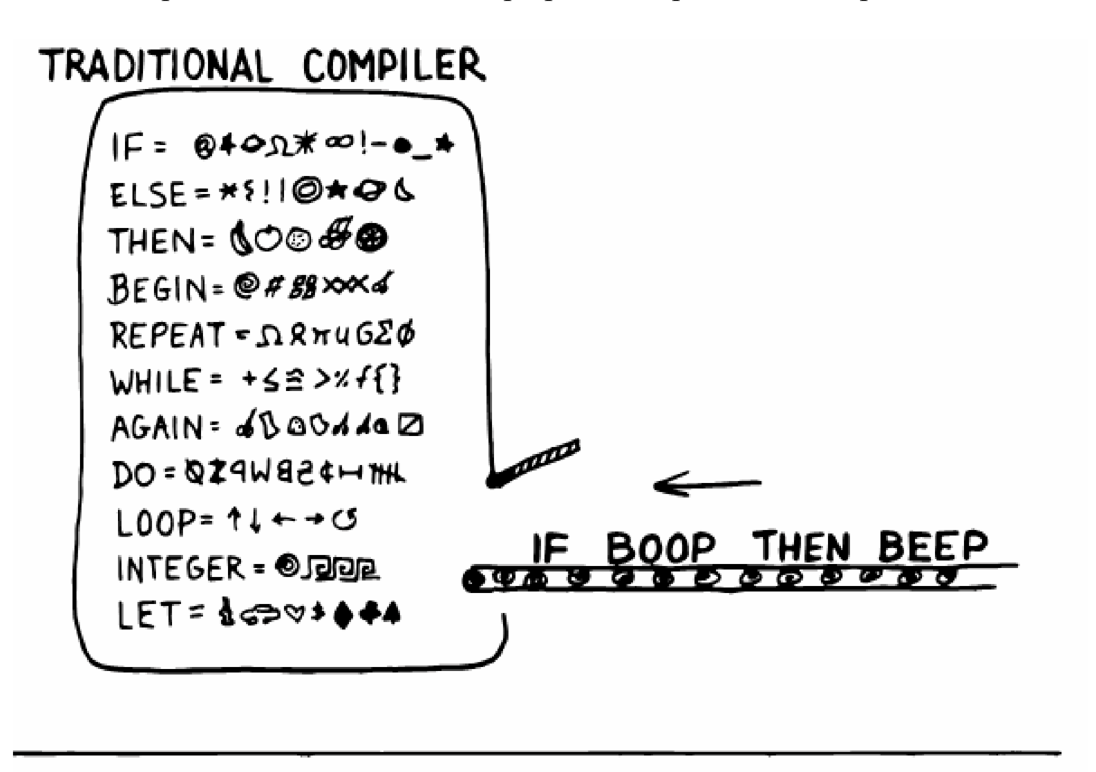
Ничего не сказано о специальных компилирующих словах типа IF, THEN, ELSE и т.д. Компилятор определения через двоеточие ничего не знает об этих словах. Он просто распознает определенные слова как "немедленные" и исполняет их, позволяя им проделывать свою собственную работу. (Смотрите "Начальный курс ", главу 11.)

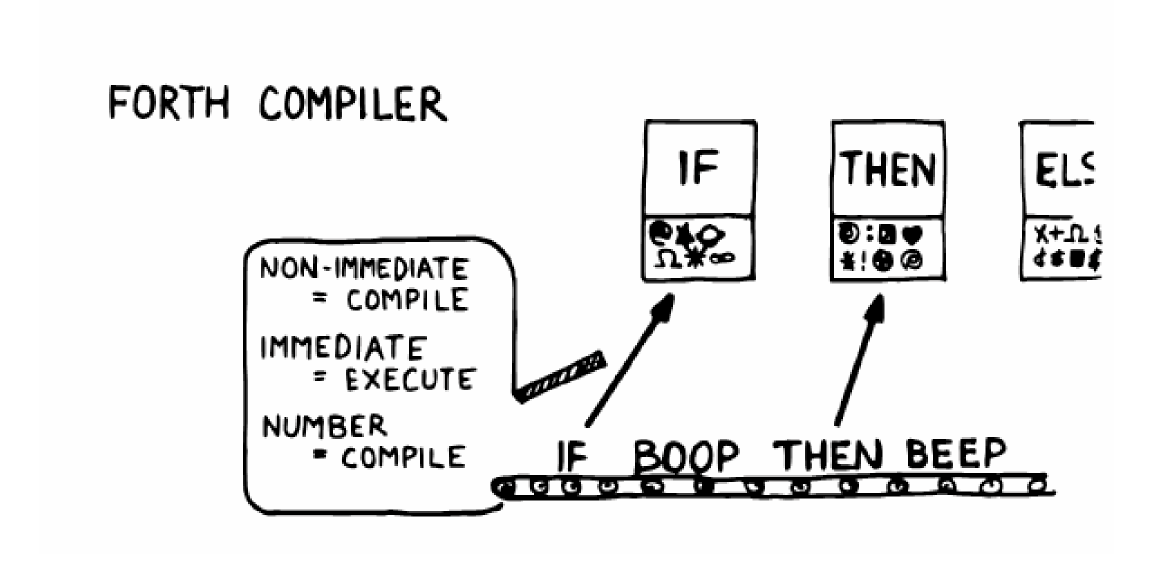
Компилятор даже не "контролирует" появление точки-запятой для завершения компиляции. Вместо этого он `исполняет` ее, позволяя ей проделать работу по завершению определения и выключению компилятора.

Имеются два огромных преимущества такого подхода. Первое, компилятор настолько прост, что может быть записан всего в несколько строк программы. Второе, нет ограничений на число компилирующих слов, которые Вы в любой момент можете добавлять, просто делая их "немедленными". Итак, даже компилятор определений Форта может наращиваться!

Текстовый интерпретатор Форта и его адресный интерпретатор также подчиняются этому правилу.

Рис.4-7. Традиционный компилятор против Форт-компилятора.





Следующий совет - быть может, самый важный в этой главе:

СОВЕТ

Не пишите свой собственный интерпретатор/компилятор, если можно использовать готовый из Форта.

Один из классов задач состоит в определении языков специального назначения - самодостаточном наборе команд для выполнения определенной работы. Примером является ассемблер машинного кода. Это большая группа мнемоник команд, с помощью которых можно описывать необходимые инструкции. И здесь подход Форта также радикально отличается от обычной философии.

Традиционные ассемблеры являются интерпретаторами специального назначения - то есть сложными программами, которые могут просматривать текст на ассемблерном языке для распознавания мнемоник типа ADD, SUB, JMP, и т.п. и собирать машинные инструкции соответственно. В то же время Форт-ассемблер - это просто лексикон Форт-слов, каждое из которых само собирает соответствующую машинную инструкцию.

Можно указать еще множество языков специального назначения. К примеру:

1. Если Вы строите приключенческую игру, Вам может захотеться написать язык, позволяющий описывать чудовищ, комнаты и т.д. Вы могли бы создать определяющее слово КОМНАТА для использования в виде:

   КОМНАТА ТЕМНИЦА

   Затем создать набор слов для описания атрибутов комнаты, строя невидимые структуры данных, связанные с комнатой:

   К-ВОСТОКУ ЛОГОВО-ДРАКОНА
   К-ЗАПАДУ МОСТ
   СОДЕРЖИТ ГОРШОК-С-ЗОЛОТОМ

   и т.д.

   Команды этого языка для построения игр могут быть просто словами Форта, с Фортом же в качестве интерпретатора.

2. Если Вы работаете с программируемыми логическими матрицами (ПЛМ), Вам может понадобиться форма описания поведения выходных сигналов в логических терминах, основанных на состояниях входных сигналов. Программатор ПЛМ был замечательно просто написан на Форте Майклом Столовицем [4].
3. Если Вы должны создать ряд пользовательских меню для управления Вашей задачей, то Вам может вначале захотеться разработать язык-компилятор меню. Слова этого нового языка позволяют программисту быстро программировать необходимые меню - при этом упрятывая информацию о том, как рисуются рамочки, двигается курсор и т.д.

Все эти примеры могут быть закодированы на Форте в качестве лексиконов, с использованием обычного Форт-интерпретатора, без необходимости написания специализированного интерпретатора или компилятора.

Мур:

Простое решение не загораживает проблему тем, что к делу не относится. Допустим, что нечто в задаче требует уникального интерпретатора. Но раз уж Вы видете такой уникальный интерпретатор, то это предполагает, что в самой

проблеме есть нечто определенно ужасное. И почти никогда в ней на самом деле этого нет.

Если Вы пишете свой собственный интерпретатор, то он почти наверняка получается самой сложной, трудоемкой частью всей задачи. Вам приходится переключаться с решения проблемы на его написание.

Мне кажется, что программисты любят писать интерпретаторы. Они обожают делать эти сложные, трудоемкие вещи. Но в конце концов наступает время, когда приходится заканчивать с программированием нажатий на клавиши или преобразованием чисел в двоичный вид и приступать к решению проблем.

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

В главе второй мы изучили способы описания задачи в терминах интерфейсов и правил. В этом разделе мы детализируем концептуальную модель для каждого из компонетов до уровня четко определенных алгоритмов и структур данных.

Алгоритм - это процедура, описанная как конечное количество правил и для выполнения определенной задачи. Правила должны быть однозначными и должны гарантированно приводить к завершению решения через конечное число шагов. (Алгоритм назван так в честь персидского математика IX века Аль-Хорезми.)

Алгоритм лежит посередине между неточными директивами человеческой речи, такими, как "Пожалуйста, рассортируйте письма в хронологическом порядке", и точными директивами компьютерных языков, таких, как "BEGIN 2DUP < IF ..." и т.д. Алгоритм хронологической сортировки писем может быть таким:

1. Взять неотсортированное письмо и посмотреть его дату.
2. Найти соответствующий скоросшиватель для этого месяца и года.
3. Пролистать письма в нем, начиная с первого, до тех пор, пока не обнаружится письмо с более поздней, чем у Вашего, датой.
4. Вложить Ваше текущее письмо перед письмом с более поздней датой. (Если скоросшиватель пуст, просто вложить письмо.)

Может быть несколько возможных алгоритмов для одной и той же работы. Приведенный выше алгоритм работал бы хорошо для скоросшивателей с десятком или меньшим количеством писем, но для скоросшивателей с сотнями писем Вам, возможно, пришлось бы обратиться к более эффективному алгоритму, такому, как:

1. (то же)
2. (то же)
3. Если дата попадает на первую половину месяца, открыть скоросшиватель на его трети. Если находящееся здесь письмо датировано позже Вашего, искать вперед до тех пор, пока не найдется письмо с той же или более ранней датой. Здесь вставить свое письмо. Если открытое письмо датировано раньше Вашего, искать назад ...

... надоело. Этот второй алгоритм сложнее первого. Однако при исполнении он потребует в среднем меньшее количество шагов (поскольку Вам не надо каждый раз начинать поиск от начала скоросшивателя) и таким образом будет проходить быстрее.

Структура данных - это собрание данных или мест для хранения этих данных, организованное специально для потребностей задачи. В последнем примере полка со

скоросшивателями и сами скоросшиватели, содержащие индивидуальные письма, могут рассматриваться как структуры данных.

Новая концептуальная модель содержит полки и скоросшиватели (структуры данных) плюс шаги для производства сортировки (алгоритмы).

РАСЧЕТЫ ИЛИ СТРУКТУРЫ ДАННЫХ ИЛИ ЛОГИКА

Мы постановили, что лучшим решением задачи является самое простое из адекватных; для каждой проблемы мы должны стремиться к самому простому подходу.

Предположим, мы должны написать код, удовлетворяющий следующим требованиям:

если входной аргумент равен 1, на выходе - 10
если входной аргумент равен 2, на выходе - 12
если входной аргумент равен 3, на выходе - 14

Мы могли бы выбрать один из трех подходов:

```
`Расчет`
( n)  1-  2*  10 +

`Структура данных`
CREATE ТАБЛИЦА  10 C,   12 C,   14 C,
( n)  1-  ТАБЛИЦА +  C@

`Логика`
( n)  CASE
    1 OF  10  ENDOF
    2 OF  12  ENDOF
    3 OF  14  ENDOF   ENDCASE
```

Для данной задачи вычисление результата - самое простое решение. Предполагая, что оно также адекватно (скорость работы некритична), принимаем, что расчет - это наилучшее из решений.

Задача преобразования углов в синусы и косинусы может быть решена более просто (по крайней мере, в терминах строк текста программы и объема объектного кода) с помощью вычисления ответов, нежели при применении структур данных. Однако для многих быстрых приложений лучше извлекать ответ из таблицы, хранимой в памяти. В этом случае простейшим `адекватным` решением является применение таблицы.

Во второй главе мы представили задачу вычисления телефонных тарифов. Эти последние казались близки к произвольным, поэтому мы спроектировали структуру данных:

	Полный тариф	Средний тариф	Низкий тариф
Первая мин.	.30	.22	.12
Дополн. Мин.	.12	.10	.06

Использование структуры данных было проще, чем попытки придумать формулу, с помощью которой эти величины могли быть вычислены. И формула могла позже оказаться неверной. В этом случае управляемый таблицей код легче сопровождать.

В третьей главе мы спроектировали интерпретатор клавиш для нашего Крошечного редактора с использование таблицы решений:

```
`Клавиша`     `        Не-вставка`   `       Вставка`
  Ctrl-D               СТЕРЕТЬ                ВЫКЛ-ВСТАВКУ
  Ctrl-I               ВКЛ-ВСТАВКУ            ВЫКЛ-ВСТАВКУ
  Забой                НАЗАД                  ВСТАВИТЬ<
```

и т.д.

Мы могли бы достигнуть того же результата с помощью логики:

```
CASE
        CNTRL-D   OF  'ВСТАВКА @  IF
                ВЫКЛ-ВСТАВКУ  ELSE  СТЕРЕТЬ     THEN  ENDOF
        CNTRL-I   OF  'ВСТАВКА @  IF
                ВЫКЛ-ВСТАВКУ  ELSE  ВКЛ-ВСТАВКУ  THEN  ENDOF
        ЗАБОЙ     OF  'ВСТАВКА @  IF
                ВСТАВКА<     ELSE  НАЗАД       THEN  ENDOF
ENDCASE
```

однако логика более запутана. И использование логики для выражения таких содержащих множество условий алгоритмов становится еще более запутанным, если таблица была применена при начальном проектировании.

Использование логики можно рекомендовать, когда результат невычисляем или когда решение не слишком сложно для применения таблицы решений. Глава 8 посвящена вопросу минимизации использования логики в Ваших программах.

СОВЕТ

При выборе того, какой подход применить к решения задачи, отдавайте свое предпочтение в следующем порядке:

1. вычисления (кроме случаев, когда существенна скорость)
2. структуры данных
3. логика

Конечно, одной из славных черт модульных языков типа Форта является то, что действительная реализация компонента - использует ли он вычисления, структуры данных или логику – не обязана быть видимой для остальной задачи.

РЕШЕНИЕ ЗАДАЧИ: ВЫЧИСЛЕНИЕ РИМСКИХ ЦИФР

В этом разделе мы попытаемся продемонстрировать процесс разработки лексикона. Вместо того, чтобы просто представлять эту задачу и ее решение, мне хотелось бы вместе с Вами с ней разобраться. (Я сохранил запись процесса своего мышления с тех пор, как решал ее впервые.) Вы увидите уже рассмотренные элементы путей решения проблем при их применении в кажущемся случайным порядке - как раз так, как это происходит в действительности.

Вот она: Задача состоит в написании определения, которое получает число со стека и отображает его в виде римской цифры.

Эта задача, вероятнее всего, представляет собой компонент большой системы. Мы, видимо, определим несколько слов при ее решении, включая структуры данных. Однако конкретно лексикон будет включать в себя только одно слово - ПО-РИМСКИ, которое будет получать аргумент со стека. (Иные слова будут внутренними для компонента.)

Мы воспользуемся научным методом - посмотрим на реальность, смоделируем решение, сверим его с реальностью, изменим решение и т.д. Начнем с воспоминаний о том, что представляют собой римские цифры.

Реально мы не помним никаких формальных правил об этих цифрах. Однако если нам дадут число, мы сумеем представить его в римской форме. Мы знаем, как это делается - но не можем пока установить алгоритм этой процедуры.

Так, давайте посмотрим на первые десять римских цифр:

I
II
III
IV
V
VI
VII
VIII
IX
X

Можно сформулировать несколько наблюдений. Первое, имеется некоторое совпадение - когда число представлено количеством символов (3 = III). С другой стороны, используются специальные символы для представления групп (5 = V). Видимо, мы не можем иметь более, чем три символа "I" в ряду до использования следующего символа.

Второе, имеется симметрия вокруг числа пять. Есть символ для пятерки (V), и символ для десятки (X). Последовательности I, II, III повторяются во второй половине, но с предшествующей V.

На-еденицу-меньше-пяти записывается как IV, а на-единицу-меньше-десяти - как IX. Похоже, что помещение I перед символом большего значения звучит как "на-еденицу-меньше-чем..."

Это смутные, беспорядочные наблюдения. Но все в порядке. Мы просто пока не имеем полной картины.

Давайте изучим, что происходит после десяти:

XI
XII
XIII
XIV
XV
XVI
XVII
XVIII
XIX
XX

Это та же последовательность, что и представленная выше, с добавленным в начале символом "X". Так что имеется также повторяющийся цикл для десяти чисел.

Если мы посмотрим на двадцатые числа, то будет то же, но с двумя "XX" в начале. Таким образом, количество "X" - то же самое, что и число в разряде десяток исходного десятичного числа.

Это выглядит как важное наблюдение: мы можем разбить наше десятичное число на десятичные цифры, и толковать каждую по отдельности. К примеру, 37 может быть записано как

XXX (тридцать)

и после него

VII (семь)

Это может оказаться преждевременным, однако мы уже видим метод, по которому Форт позволяет нам разбить число на десятичные цифры - делением с остатком на десять. К примеру, если мы напишем

```
37 10 /MOD
```

то получим 7 и 3 на стеке (три - как частное - на вершине).

Однако при этом возникает вопрос: как насчет чисел, меньших десяти? Не исключительный ли это случай? Ну, если мы предполагаем, что каждый "X" представляет десятку, то отсутствие "X" представляет ноль. Это - `не` исключительный случай. Наш алгоритм работает даже при числах, меньших десяти.

Давайте продолжим наблюдения, уделяя особое внимание циклам десяти. Мы отмечаем, что сорок будет "XL". Это аналогично тому, как 4 является "IV", только сдвинутое на десять. "X" перед "L" говорит "на-десять-меньше-пятидесяти". Точно так же,

L (50)	аналогично	V (5)
LX (60)		VI (6)
LXX (70)		VII (7)
LXXX (80)		VIII (8)
XC (90)		IX (9)
C (100)		X (10)

Те же самые последовательности применимы к любым десятичным цифрам - меняются только сами символы. В любом случае ясно, что мы имеем дело с существенно десятичной системой.

Если было бы нужно, мы даже смогли бы написать модель системы для показа римских цифр от 1 до 99, использующую комбинацию алгоритма со структурой данных.

Структура данных:

Таблица единиц		Таблица десятков	
0		0	
1	I	1	X
2	II	2	XX
3	III	3	XXX
4	IV	4	XL
5	V	5	L

6	VI	6	LX
7	VII	7	LXX
8	VIII	8	LXXX
9	IX	9	XC

Алгоритм:

Разделить на 10. Частное является числом десятков; остаток - это число единиц. Найти строку для десятков в таблице десятков и напечатать соответствующую последовательность символов. Найти строку для единиц в таблице единиц и напечатать соответствующую последовательность символов.

К примеру, если число равно 72, частное равно 7, остаток - 2. 7 в таблице десятков соответствует "LXX", печатаем это. 2 в таблице единиц дает "II", так что печатаем эти символы.

Результат таков:

LXXII

Мы сконструировали модель, которая работает для чисел от одного до 99. Любое большее число потребует таблицу сотен при начальном делении на 100.

Описанная логическая модель может оказаться удовлетворительной, поскольку выполняет свои функции. Однако как-то не видно, что мы полностью разрешили проблему. Мы не выделяли вопрос о том, как получить основную последовательность, записав все возможные комбинации в многочисленных таблицах. Ранее в данной главе упоминалось о том, что вычисление ответа, если оно возможно, может быть более легким, чем использование структур данных.

Поскольку мы в этой секции изобретаем алгоритмы, давайте проделаем весь путь до конца. Поищем общий алгоритм для составления произвольного числа, с использованием лишь элементарного набора символов. Наша структура данных должна была бы содержать лишь следующую толику информации:

I	V
X	L
C	D
M	

При составлении этой таблицы мы также `организовали` символы так, как это кажется правильным. Символы в левой колонке кратны десяти; символы в правой кратны пяти. Далее, символы в каждом ряду ровно в десять раз отличаются от символов над ними.

Другое различие состоит в том, что все символы в первой колонке могут быть скомбинированы, например "XXXIII". Однако нельзя объединять символы в правой колонке, типа "VVV". Полезно ли это наблюдение? Как знать?

Условимся называть символы в первой колонке ЕДИНИЧКИ, а в правой - ПЯТЕРКИ. ЕДИНИЧКИ представляют значения 1, 10, 100 и 1000, то есть значения всех возможных десятичных позиций. ПЯТЕРКИ представляют 5, 50 и 500, то есть значения пятерок во всех возможных десятичных позициях.

Используя эти термины вместо самих символов, мы смогли бы выразить алгоритм для представления любых чисел. (Мы ведь выделили настоящие символы из `подобия` символов.) В частности, мы можем сформулировать следующий начальный алгоритм:

Для любой цифры напечатать столько символов из ЕДИНИЧКИ, сколько необходимо для суммы числа.

Так, для 300 получаем "CCC", для 20 имеем "XX", для одного - "I". И для 321 выходит "CCCXXI".

Такой алгоритм работает вплоть до цифры 4. Теперь давайте расширим его для следующего исключения:

Печатать столько ЕДИНИЧЕК, сколько необходимо для суммы числа, однако если цифра равна 4, напечатать сначала ЕДИНИЧКУ, затем ПЯТЕРКУ.

Таким образом, 40 - это "XL"; 4 - это "IV".

Новый вариант правила работает до числа 5. Как мы заметили раньше, цифры от пяти и выше начинаются с ПЯТЕРКИ. Вновь расширяем наше правило:

Если цифра больше 5, начинать с ПЯТЕРКИ и вычитать пять из числа; иначе этого не делать. Затем печатать столько из ЕДИНИЧКИ, сколько необходимо для суммы числа. Однако если число равно 4, печатать только ЕДИНИЧКУ и ПЯТЕРКУ.

Такое правило работает до цифры 9. В этом случае мы должны напечатать ЕДИНИЧКУ до - чего? До ЕДИНИЧКИ из следующего десятичного уровня (в ряду под ним). Давайте назовем ее ДЕСЯТКОЙ. Полная наша модель тогда будет:

Если цифра - 5 или более, начинать с ПЯТЕРКИ и вычитать пять из числа; иначе - не делать этого. Затем печатать столько ЕДИНИЧЕК, сколько необходимо для суммы числа. Однако если цифра равна 4, печатать только ЕДИНИЧКУ и ПЯТЕРКУ, а если цифра равна 9, печатать только ЕДИНИЧКУ и ДЕСЯТКУ.

Теперь у нас есть русскоязычная версия алгоритма. Однако осталось еще несколько шагов, которые надо сделать прежде, чем мы сможем запустить эту версию на компьютере.

А именно, нам следует уточниться насчет исключений. Нельзя просто сказать

Сделать А, Б и В. `Но` в таких-то и таких-то случаях сделать что-то другое, поскольку ЭВМ сделает А, Б и В до того, как рассмотрит их получше.

Вместо этого нам следует проверить наличие исключений `до` того, как что-нибудь делать.

СОВЕТ

При прокладывании алгоритма вначале примите во внимание исключительные случаи. При написании кода вначале опишите исключения.

Это дает нам дополнительное представление об общей структуре нашего числопечатающего слова. Слово должно начинаться с проверки исключений на 4/9. В каждом из таких случаев оно должно сработать соответственно. Если ни одно из исключений не появляется, оно должно выполнять "нормальный" алгоритм. На псевдокоде:

```
: ЦИФРА  ( n )  4-ИЛИ-9?  IF  особые случаи
```

```
        ELSE  обычный случай  THEN ;
```

Опытный Форт-программист не выписывал бы в действительности этот псевдокод, но скорее сформировал бы мысленное представление о способе устранения особых случаев. Менее опытному могли бы оказаться полезными рисунки структуры в виде диаграмм или в коде, как мы сделали это здесь.

Мы пытаемся минимизировать зависимость программы на Форте от логики. Однако в этом случае нам нужен условный оператор IF, поскольку имеются исключительные ситуации, которые нужно учитывать. Впрочем, мы все равно минимизировали сложность управляющей структуры, ограничив число конструкций IF THEN в данном определении до одной.

Да, нам все еще приходится различать случай 4-х от случая 9-ти, однако мы перепоручили это структурное определение определению более низкого уровня - проверку "4-или-9" и обработку "особого случая".

Что в действительности ясно из нашего определения, так это то, что `любое` из исключений - 4 или 9 - должно запрещать исполнение обычного случая. Недостаточно просто проверять на каждое из исключений, как в такой версии:

```
: ЦИФРА  ( n)  4-СЛУЧАЙ?  IF  ЕДИНИЧКА ПЯТЕРКА   THEN
9-СЛУЧАЙ?  IF  ЕДИНИЧКА ДЕСЯТКА   THEN
        обычный случай... ;
```

поскольку без обычного случая никогда не обходится. (Нет способа поместить ELSE перед обычным случаем, поскольку часть ELSE должна находиться между IF и THEN.)

Если мы настаиваем на раздельной обработке обоих исключений, то должны сформировать в каждом из них передачу дополнительного флага о том, что исключение возникло. Если один из этих флагов установлен, тогда обычный случай исключается:

```
: ЦИФРА  ( n)  4-СЛУЧАЙ?  IF  ЕДИНИЧКА ПЯТЕРКА   THEN
        9-СЛУЧАЙ?  IF  ЕДИНИЧКА ДЕСЯТКА   THEN
        OR  NOT  IF  обычный случай  THEN ;
```

Однако этот подход неоправданно усложняет определение, добавляя новые структуры управления. Оставим все как было.

Теперь у нас есть общая идея о структуре нашего главного определения.

Мы сказали: "если цифра - 5 или более, начинать с ПЯТЕРКИ и вычитать пять из числа; иначе ничего не делать. Затем напечатать столько ЕДИНИЧЕК, сколько необходимо для суммы числа".

Прямое преобразование этих правил в Форт может выглядеть как:

```
( n)  DUP  4 > IF  ПЯТЕРКА  5 -  THEN  ЕДИНИЧКИ
```

Такая запись технически корректна, однако мы знакомы с техникой деления по модулю, а ведь это - типовая ситуация для использования деления по модулю 5. Если мы поделим число на пять, частное будет нулем (логическая ложь) когда число меньше пяти и единицей (истина) если оно находится между 5 и 9. Мы можем использовать это как булевский флаг для определения того, нужна ли ПЯТЕРКА:

```
( n)  5 /  IF  ПЯТЕРКА  THEN ...
```

Частное-флаг становится аргументом для IF.

Далее, остаток от деления на 5 всегда представляет собой число от 0 до 4, что означает возможность (кроме случая исключений) использования остатка непосредственно в качестве аргумента для единичек. Мы перепишем фразу в виде:

```
( n)  5 /MOD  IF  ПЯТЕРКА  THEN   ЕДИНИЧКИ
```

Возвращаясь к нашим исключениям, мы теперь отмечаем, что на 4 и на 9 можно проверить в единственном месте - а именно, если остаток равен 4. Очевидно, что мы можем сделать наше 5 /MOD вначале, после чего проверить на исключения. Что-то вроде:

```
: ЦИФРА  ( n)
        5 /MOD  OVER 4 =  IF  особый случай  ELSE
        IF  ПЯТЕРКА  THEN   ЕДИНИЧКИ   THEN ;
```

(Отметьте, что мы проделали OVER с остатком, поэтому не потеряли его при сравнении с 4.)

Выходит, что мы в конце концов `сделали` IF THEN двойного вложения. Однако это, кажется, правильно, поскольку такой IF THEN обслуживает особый случай. Другой же IF THEN – столь короткая фраза, как "IF ПЯТЕРКА THEN", вряд ли заслуживает того, чтобы делать под нее отдельное определение. Вы, впрочем, можете. (Но мы не будем.)

Давайте сфокусируемся на коде для особых случаев. Вот его алгоритм: "Если цифра - четыре, то напечатать ЕДИНИЧКУ и ПЯТЕРКУ. Для девятки дать ЕДИНИЧКУ и ДЕСЯТКУ".

Можно положить, что цифра обязательно будет либо той, либо другой, иначе мы бы это определение не исполняли. Вопрос в том, как узнать, которая?

Опять же, можно использовать остаток от деления на пять. Если он равен нулю, то цифра была четверкой; иначе - девяткой. Так что мы разыграем ту же партию и используем частное как булевский флаг. Напишем:

```
: ПОЧТИ  ( частное )
        IF  ЕДИНИЧКА ДЕСЯТКА  ELSE  ЕДИНИЧКА ПЯТЕРКА  THEN ;
```

При повторном взгляде отметим, что ЕДИНИЧКУ мы исполняем в любом случае. Можно упростить определение до:

```
: ПОЧТИ  ( частное )
    ЕДИНИЧКА IF  ДЕСЯТКА  ELSE  ПЯТЕРКА  THEN ;
```

Мы считаем, что на стеке у нас имеется частное. Давайте вернемся к определению ЦИФРА и подумаем о том, что там происходит:

```
: ЦИФРА  ( n)
    5 /MOD  OVER 4 =  IF  ПОЧТИ  ELSE
    IF  ПЯТЕРКА  THEN   ЕДИНИЧКИ   THEN ;
```

Выходит, что у нас имеется не только частное, но под ним и остаток. Мы их оба храним на стеке для случая перехода на часть ELSE. Слово ПОЧТИ, однако, употребляет только частное. Поэтому для сохранения симметрии мы обязаны написать для остатка DROP:

```
: ЦИФРА  ( n)
```

```
5 /MOD  OVER 4 =  IF  ПОЧТИ DROP  ELSE
IF  ПЯТЕРКА  THEN   ЕДИНИЧКИ   THEN ;
```

У нас получилось полное определение в коде для печати одной римской цифры. Если бы мы спешили срочно проверить его до написания необходимых внешних определений, то могли бы легко набросать лексикон слов, нужных для печати группы символов, скажем, ЕДИНИЧЕК:

```
: ЕДИНИЧКА    ." I" ;
: ПЯТЕРКА     ." V" ;
: ДЕСЯТКА     ." X" ;
: ЕДИНИЧКИ   ( #-единичек)
     ?DUP IF  0 DO  ЕДИНИЧКА  LOOP  THEN ;
```

до загрузки наших слов ПОЧТИ и ЦИФРА.

Но мы не столь нетерпеливы. Нет, мы озабочены тем, чтобы слова ЕДИНИЧКА, ПЯТЕРКА и ДЕСЯТКА давали символы, зависящие от положения цифры в числе.

Давайте вернемся к таблице символов, которую рисовали раньше:

	единицы	пятёрки
единицы	I	V
десятки	X	L
сотни	C	D
тысячи	M	

Мы встретились также с необходимостью иметь "ДЕСЯТКИ" - т.е. ЕДИНИЧКИ на один ряд ниже. Как если бы на самом деле написать таблицу в виде:

	единицы	пятёрки	десятки
единицы	I	V	X
десятки	X	L	C
сотни	C	D	M
тысячи	M		

Однако это кажется избыточным. Нельзя ли избежать этого? Может быть, попытаться нарисовать другую таблицу, например, линейную:

Единицы	I
	V
Десятки	X
	L
Сотни	C
	D
Тысячи	M

Можно представить себе, что имя каждого заголовка ("единицы", "десятки" и др.) указывает на позицию соответствующей ЕДИНИЧКИ. Отсюда можно также получить

ПЯТЕРКУ, опускаясь на одну строку ниже текущей ЕДИНИЧКИ, и ДЕСЯТКУ, опускаясь на две строки.

Это похоже на создание указателя с тремя стрелками. Мы его можем подсоединить к ряду единиц, как на рис. 4-8б, или к любой другой десятичной степени.

Рис.4-8. Механистическое представление доступа к структуре данных.



Опытный фортист вряд ли воображает себе такие указатели или что-нибудь в этом духе. Однако сильный внутренний образ должен присутствовать - как основа для правильного хода мысли – до того, как будет предпринята попытка перенести модель в код.

Новичкам, развивающим в себе правильный метод мышления, может оказаться полезным такое замечание:

СОВЕТ

Если у Вас возникают проблемы с представлением
концептуальной модели - визуализируйте ее, т.е. нарисуйте,
в виде механического приспособления.

Наша таблица - это просто массив символов. Поскольку на каждый из них нужен всего байт, то давайте в качестве одной "позиции" примем этот один байт. Таблицу мы назовем РИМСКИЕ (*):

```
CREATE РИМСКИЕ    ( единицы)  ASCII I  C,   ASCII V  C,
          ( десятки)      ASCII X  C,   ASCII L  C,
          ( сотни)        ASCII C  C,   ASCII D  C,
          ( тысячи)       ASCII M  C,
```

(*) - В некоторый системах применение байтовой компиляции и слова C, запрещено. В этом случае используйте слово , и размер "позиции" в таблице, равный 2-м байтам.

Примечание: такое использование слова ASCII требует, чтобы оно было определено как зависящее от STATE (см. приложение B). Если у Вас слова ASCII нет или оно определено по-иному, пишите:

```
CREATE РИМСКИЕ   73 C,  86 C,  88 C,  76 C,
          67 C,  68 C,  77 C,
```

Нужный символ из таблицы можно выбрать, одновременно применяя два различных смещения. Одно измерение представляет десятичное место: единицы, десятки, сотни и т.д. Его устанавливают "текущим", и оно сохраняет свое состояние до тех пор, пока мы его не изменим.

Другое измерение выбирает желаемый тип символа - ЕДИНИЧКУ, ПЯТЕРКУ или ДЕСЯТКУ - внутри текущей десятичной позиции. Это измерение - случайное, то есть мы каждый раз указываем, какой из символов хотим получить.

Давайте начнем с создания "текущего" измерения. Нам нужен способ для указания текущей десятичной точки отсчета. Создадим переменную по имени #ПОЗИЦИИ (произносится "номер-позиции") и пусть в ней хранится смещение в таблице:

```
VARIABLE #ПОЗИЦИИ
: ЕДИНИЦЫ     0 #ПОЗИЦИИ ! ;
: ДЕСЯТКИ     2 #ПОЗИЦИИ ! ;
: СОТНИ       4 #ПОЗИЦИИ ! ;
: ТЫСЯЧИ      6 #ПОЗИЦИИ ! ;
```

Теперь можно изыскать путь для задания положения "стрелки" - добавляя содержимое #ПОЗИЦИИ к начальному адресу таблицы, оставляемому словом РИМСКИЕ:

```
: ПОЗИЦИЯ  ( -- адр-позиции )  РИМСКИЕ  #ПОЗИЦИИ @ + ;
```

Посмотрим, нельзя ли реализовать одно из слов для печати символа. Начнем с ЕДИНИЧКИ.

Мы хотим, чтобы оно выдавало (через EMIT) символ.

```
: ЕДИНИЧКА                    EMIT ;
```

Работая назад, обнаруживаем, что слово EMIT требует наличия на стеке кода ASCII символа. Откуда он там возьмется? С помощью слова C@.

```
: ЕДИНИЧКА            C@ EMIT ;
```

Слово же C@ требует `адрес` ячейки, которая содержит нужный символ. Как мы его получим?

ЕДИНИЧКА - это первая "стрелка" перемещаемого указателя - позиция, на которую сразу и показывает слово ПОЗИЦИЯ. Поэтому нужный нам адрес получается просто:

```
: ЕДИНИЧКА    ПОЗИЦИЯ   C@ EMIT ;
```

Теперь давайте напишем слово ПЯТЕРКА. Оно вычисляет тот же адрес ячейки, но потом добавляет к нему единицу для перемещения к следующей ячейки перед получением символа:

```
: ЕДИНИЧКА    ПОЗИЦИЯ 1+  C@ EMIT ;
```

А ДЕСЯТКА получается так:

```
: ЕДИНИЧКА    ПОЗИЦИЯ 2+  C@ EMIT ;
```

Три эти определения избыточны. Поскольку единственным различием между ними является смещение, то можно выделить смещение из остальных определений:

```
: .СИМВОЛ  ( смещение)  ПОЗИЦИЯ +  C@ EMIT ;
```

Теперь можно определить:

```
: ЕДИНИЧКА   0 .СИМВОЛ ;
: ПЯТЕРКА    1 .СИМВОЛ ;
: ДЕСЯТКА    2 .СИМВОЛ ;
```

Все, что нам остается - это разбить наше полное десятичное число на последовательность десятичных цифр. Благодаря уже сделанным наблюдениям это сделать несложно. На рисунке 4-9 показан наш законченный вариант.

Ура! От проблемы, через концептуальную модель - и к коду.

Замечание: это решение не оптимально. Данная книга не рассматривает фазу оптимизации.

Еще одно соображение: В зависимости от того, где используется эта задача, нам может понадобиться проверка на ошибочные ситуации. Действительно, максимальная известная нам цифра - это M, и самое большое число, которое мы способны представить - это 3999 или MMMCMXCIX.

ПО-РИМСКИ можно было бы переопределить таким образом:

```
: ПО-РИМСКИ  ( n)
     DUP  3999 > ABORT" Слишком велико"  ПО-РИМСКИ ;
```

Мур:

Когда все сделано правильно, появляется вполне определенное ощущение этой правоты. Быть может, такое ощущение и отличает Форт от других языков, в которых никогда не почувствуешь, что действительно все сделано как надо. В Форте восклицаешь "Ага!", и хочется побежать и кому-нибудь об этом рассказать. Разумеется, никто другой не воспримет это так же, как Вы.

Рис.4-9. Решение задачи о римских цифрах.

Блок # 20

```
\ Римские цифры                                                8/18/83
CREATE РИМСКИЕ   ( единицы)  ASCII I  C,  ASCII V  C,
        ( десятки)     ASCII X  C,  ASCII L  C,
        ( сотни)       ASCII C  C,  ASCII D  C,
        ( тысячи)      ASCII M  C,
VARIABLE #ПОЗИЦИИ
: ЕДИНИЦЫ        0 #ПОЗИЦИИ ! ;
: ДЕСЯТКИ        2 #ПОЗИЦИИ ! ;
: СОТНИ          4 #ПОЗИЦИИ ! ;
: ТЫСЯЧИ         6 #ПОЗИЦИИ ! ;

: ПОЗИЦИЯ  ( -- адр-позиции)  РИМСКИЕ  #ПОЗИЦИИ @  + ;
```

Блок # 21

```
\ Римские цифры ...                         8/18/83
: .СИМВОЛ  ( смещение)  ПОЗИЦИЯ +  C@ EMIT ;
: ЕДИНИЧКА       0 .СИМВОЛ ;
: ПЯТЕРКА        1 .СИМВОЛ ;
: ДЕСЯТКА        2 .СИМВОЛ ;

: ЕДИНИЧКИ  ( #-единичек -- )
        ?DUP  IF  0 DO  ЕДИНИЧКА  LOOP  THEN ;
: ПОЧТИ  ( частное-от-5-/ -- )
        ЕДИНИЧКА  IF  ДЕСЯТКА  ELSE  ПЯТЕРКА  THEN ;
: ЦИФРА  ( цифра -- )
        5 /MOD  OVER 4 = IF  ПОЧТИ DROP  ELSE  IF  ПЯТЕРКА  THEN
        ЕДИНИЧКИ  THEN ;
```

Блок # 22

```
\ Римские цифры ...                                            8/18/83
: ПО-РИМСКИ  ( число -- )  1000 /MOD          ТЫСЯЧИ ЦИФРА
                       100 /MOD               СОТНИ ЦИФРА
                       10 /MOD                ДЕСЯТКИ ЦИФРА
```

4 ЕДИНИЦЫ ЦИФРА ;
5

ИТОГИ

В этой главе мы научились разрабатывать отдельные компоненты, начав с размышлений над их синтаксисом, затем продолжив определением их алгоритма(ов) и структур данных, и завершив все реализацией на Форте.

Этой главой мы завершаем обсуждение стадии проектирования. В оставшейся части книги мы рассмотрим стиль и технику программирования.

ЛИТЕРАТУРА

1. G. Polya. `How To Solve It: A New Aspect of Mathematical Method,` (Princeton, New Jersey, Princeton University Press).
2. Leslie A. Hart, `How the Brain Works,` (c) 1975 by Leslie A. Hart, (New York, Basic Books, Inc., 1975).
3. Evan Rosen, "High Speed, Low Memory Consuption Structures," `1982 FORML Conference Proceedings,` p.191.
4. Michael Stolowitz, "A Compiler for Programmable Logic in FORTH," `1982 FORML Conference Proceedings,` p.257.

ДЛЯ ДАЛЬНЕЙШИХ РАЗМЫШЛЕНИЙ

Спроектируйте компонент и опишите алгоритм(ы), необходимые для имитации тасования колоды карт. Этот алгоритм должен давать массив чисел от 0 до 51, сформированный случайным образом.

Разумеется, специальным ограничением здесь является то, что ни одна карта не может встречаться в колоде дважды.

Можно положить, что у Вас имеется генератор случайных чисел по имени CHOOSE. Он берет со стека аргумент "n" и дает случайное число в диапазоне от нуля до n-1 включительно. (См. "Генератор бессмысленных сообщений", глава 10 из "Начального курса...")

Сможете ли Вы спроектировать алгоритм перетасовки так, чтобы избежать утомительных проверок неопределенного количества ячеек памяти при каждом прохождении цикла? Можете ли Вы этого достичь, используя всего один массив?

ГЛАВА 5

РАЗРАБОТКА :

ЭЛЕМЕНТЫ ФОРТ - СТИЛЯ

Плохо написанная на Форте программа обычно выглядит как "код, пропущенный через пресс для мусора". Форт в действительности дает много свободы в выборе способа написания программ. Такая свобода обеспечивает все возможности для создания чрезвычайно удобочитаемого и легко управляемого кода, что достигается сознательным применением элементов хорошего Форт-стиля.

В этой главе мы просмотрим соглашения по стилистике Форта, в том числе такие вещи, как:

организация текстов программы (листингов);
планировка экранов (блоков), отступы и выравнивания;
комментирование;
выбор имен.

Я думаю, что мог бы порекомендовать всем список жестких и точных соглашений. К несчастью, такой список мог бы и не подойти во многих ситуациях. В этой главе описание множества широко известных соглашений перемежается с изложением индивидуальных привязанностей, выдвижением альтернативных идей и рассказами о причинах различных предпочтений.
Другими словами:

```
: СОВЕТ        ВЗЕШИВАТЬ СУЖДЕНИЯ ;
```

Мне хотелось бы особенно поблагодарить Кима Харриса, предложившего многие из приведенных здесь соглашений, за его непрерывные усилия в приведении к общему знаменателю разнообразных взглядов на хорошую Форт-стилистику.

ОРГАНИЗАЦИЯ ЛИСТИНГОВ

Хорошо организованная книга разбита на ясно обозначенные главы, с четко очерченными разделами и списком содержимого, что помогает охватить ее структуру единым взглядом. Хорошо организованную книгу легко читать. Плохо же огранизованная книга затрудняет понимание и делает последующий поиск информации почти невозможным.

Точно так же необходимость хорошей огранизации относится и к листингу программы. Вот три кита такой организации:

1. Декомпозиция
2. Составление
3. Распределение дискового пространства

ДЕКОМПОЗИЦИЯ.

Как мы уже видели, организация текста программы должна следовать разбиению задачи на лексиконы. В общем случае эти лексиконы надо выстраивать в порядке взаимного `использования`. Лексиконы, которые `используются`, должны предшествовать тем, которые их `используют`.

В целом элементы листинга должны быть организованы в порядке возрастания сложности, при этом самые сложные построения - появляться ближе к концу. Лучше всего группировать все так, чтобы можно было не подключать близлежащие блоки (т.е. не загружать их), и все равно иметь самодостаточную, работоспособную задачу, которая не имеет только лишь некоторых сложных возможностей.

Мы подробно рассмотрели искусство декомпозиции в главе третьей.

СОСТАВЛЕНИЕ.

Составление (композиция) состоит в складывании друг с другом кусочков для создания единого целого. Хорошая композиция требует столько же артистизма, сколько и хорошая декомпозиция.

Одно из Фортовских соглашений состоит в том, что исходные тексты размещаются в "блоках" (*), которые являются порциями массовой памяти по 1К каждый. В Форте можно сцеплять каждый блок исходного текста со следующим, получая весь листинг линейным, в виде некоего длинного пергаментного свитка. Такой подход не хорош. Вместо него:

(*) - в оригинале использовался термин "экран" ("screen").

В соответствии с более современными соглашениями здесь и далее "экраны" заменены на "блоки" ("blocks") (термин "экран" применялся ранее для обозначения блока, содержащего исходный текст).

СОВЕТ
Стройте текст Вашей программы как книгу: иерархически.

Задача может состоять из:

`Блоков:` мельчайших частей текста на Форте;

`Лексиконов:` от одного до трех блоков, достаточных для размещения компонента;

`Глав:` серий родственных лексиконов;

`Загрузочных блоков:` аналогичных оглавлению, такой блок загружает главы в нужной последовательности.

БЛОК ЗАГРУЗКИ ПРОГРАММЫ.

Рисунок 5-1 - это пример загрузочного блока. Поскольку этот блок имеет номер 1, можно загрузить всю задачу, введя

```
1 LOAD
```

Отдельные команды LOAD внутри блока загружают главы задачи. К примеру, блок 12 - загрузочный для главы видео-примитивов.

Рис.5-1. Пример загрузочного блока.

```
Блок # 1
\ QTF+  Загрузочный Блок                    07/09/83
: RELEASE#   ." 2.01" ;
  9 LOAD  \ инструментарий компилятора, примитивы языка
 12 LOAD  \ видео-примитивы
 21 LOAD  \ редактор
 39 LOAD  \ отображение строки
 48 LOAD  \ форматтер
 69 LOAD  \ окна
 81 LOAD  \ предопределения
 90 LOAD  \ обрамление
 96 LOAD  \ надписи, рисунки, таблицы
102 LOAD  \ генератор оглавления
```

В блоке загрузки программы говорится о том, где найти все ее главы. Так, если Вам захочется посмотреть на программы для обрамления, можете найти их в секции, начинающейся с блока 90.

Каждый из блоков загрузки главы, в свою очередь, загружает все блоки, входящие в эту главу. Мы вкратце изучим некоторые форматы блоков загрузки глав.

Первый выигрыш от такой иерархической схемы состоит в том, что можно загружать любой участок или любой блок поодиночке, без необходимости компиляции всей задачи. Модульность исходного текста является одной из причин скорости Фортовского цикла редактирования, загрузки и проверки (необходимых для итеративного подхода). Как и страницы книги, каждый из блоков может быть доступен индивидуально и быстро. То есть имеет место "произвольный доступ" к управлению исходным текстом.

Можно также заменять любой пассаж кода новой, испытанной версией с помощью простой замены чисел в блоке загрузки. Нет нужды в передвижении протяженных отрезков исходного текста внутри файла.

В маленьких задачах может не быть таких вещей, как главы. Блок управления загрузкой задачи будет напрямую загружать все лексиконы. В то же время в более крупных задачах дополнительный уровень иерархии позволяет улучшить управляемость программы.

Блок должен быть либо загрузочным, либо содержать программу, но не быть смешанным. Избегайте внедрения команд LOAD и THRU в середину блока с определениями только потому, что Вам "что-то нужно" или поскольку у Вас "не хватает места".

КОМАНДЫ ОБХОДА (SKIP).

Две команды облегчают управление тем, что загружается, а что игнорируется в блоке. Вот они:

\
\S (иногда ;S , а также EXIT)

\ произносится "пропустить-строку". Команда вызывает игнорирование интерпретатором Форта всего, что находится справа от нее в той же строке. (Поскольку \ является словом Форта, после него должен быть пробел.) Слову \ не требуется символ-ограничитель.

На рисунке 5-1 слово \ используется двумя способами: в начале строки-комментария (строки 0) блока и для начала комментариев в отдельных строках, в которых справа больше нет кода.

При отладке слово \ также служит для "закомментирования" строк, уже содержащих закрывающую круглую скобку в имени или комментарии. К примеру, два таких значка пропуска строки предохраняют от компиляции определение ОРЕХИ без возникновения проблем в деле учета закрывающих скобок:

```
\ : ОРЕХИ  ( x y z)
\    SWAP ROT  (ОРЕХИ) ;
```

\S произносится как "пропустить-страницу". Оно заставляет интерпретатор Форта полностью прекратить работу с текущим блоком, как будто в этом блоке больше ничего нет.

Во многих Форт-системах эта функция по действию аналогична слову EXIT, которое является программой времени исполнения для слова ;. В этих системах возможно использование слова EXIT. В то же время некоторые Форт-системы по внутренним причинам требуют другого определения для функции "пропустить-страницу".

Исходные тексты для \ и \S можно найти в приложении В.

БЛОКИ ЗАГРУЗКИ ГЛАВ.

На рисунке 5-2 продемонстрирован типичный блок загрузки главы. Загружаемые им блоки имеют относительный отсчет, а не абсолютный, как в блоке загрузки задачи.

Рис.5-2. Пример блока загрузки главы.

```
Блок # 100
    0 \ Графика              Загрузка главы        07/11/83
    1
    2  1 FH LOAD          \ примитив рисования точки
    3  2 FH 3 FH THRU     \ примитивы рисования линий
    4  4 FH 7 FH THRU     \ масштабирование, вращение
    5  8 FH LOAD          \ прямоугольник
    6  9 FH 11 FH THRU    \ круг
    7
    8
    9
    10 УГОЛ  \ инициализация относительной позиции в нижний левый
    11       \ угол
```

Это сделано потому, что такой блок является первым в последовательном наборе блоков этой главы. Вы можете перемещать всю эту главу вперед и назад по листингу; относительные указатели в блоке загрузки главы не зависят от конкретной позиции. Все, что Вам требуется поменять - это одно число в блоке загрузки задачи так, чтобы оно указывало на начало данной главы.

СОВЕТ
Используйте абсолютные номера блоков в блоке загрузки задачи. Используйте относительные номера в блоках загрузки глав или разделов.

Есть два способа реализации относительной загрузки. Наиболее часто определяют:

```
: +LOAD  ( смещение -- )  BLK @ +  LOAD ;
```

и

```
: +THRU  ( смещение-от смещение-до -- )
1+ SWAP DO  I +LOAD  LOOP ;
```

Лично мне кажется более целесообразным иной путь, при котором требуется только одно слово, FH (его определение смотрите в приложении B).

Фраза

```
1 FH LOAD
```

читается как "1-й отсюда (From Here) загрузить" и эквивалентно 1 +LOAD.

Точно так же

```
2 FH  5 FH THRU
```

читается как "от 2-го отсюда до 5-го отсюда сквозь".

Некоторые программисты начинают каждую главу с подставного слова типа

```
: ВИДЕО-ПРИМИТИВЫ ;
```

и указывают имя этого слова в строке комментария блока загрузки задачи, из которого загружается данная глава. Это позволяет выполнять FORGET выборочно для любой главы и перезагружаться с этой точки без необходимости просмотра текста этой главы.

Внутри главы первая группа блоков обычно будет определять те переменные, константы и другие структуры данных, которые будут нужны по всей главе. Следование такому принципу приводит к появлению цепи лексиконов, загружаемых в порядке "использования". Последние строки в блоке загрузки главы обычно содержат необходимые команды инициализации.

Некоторые из наиболее озабоченных стилистикой Фортовых писателей начинают каждую главу с "преамбулы", в которой в общих словах обсуждаются основы работы компонентов этой главы. Рисунок 5-3 - это пример блоков-преамбул, показывающих формат, принятый на фирме Moore Products Co.

Рис.5-3. Формат преамбулы главы в фирме Moore Products Co.

```
CHAPTER 5  -  ORIGIN/DESTINATION  - MULTILOOP BIT ROUTINES

DOCUMENTS - CONSOLE STRUCTURE CONFIGURATION
        DESIGN SPECIFICATION
        SECTIONS - 3.2.7.5.4.1.2.8
                   3.2.7.5.4.1.2.10

ABSTRACT  -  File control types E M T Q and R can all
             originate from a Regional Satellite or a
             Data Survey Satellite.  These routines allow
               the operator to determine whether the control
               originated from a Regional Satellite or not.
```

```
CHAPTER NOTES  - Whether or not a point originates from
                   a Regional Satellite is determined by
                   the Regional bit in BITS, as follows:

                   1 = Regional Satellite
                   2 = Data Survey Satellite

                   For the location of the Regional bit
                   in BITS, see the Design Specification
                   Section - 3.2.7.5.4.1.2.10

  HISTORY  -
```

Чарльз Мур (Charles Moore - не имеет никакого отношения к фирме Moore Products Co.) уделяет меньше, чем я, внимания хорошо организованному иерархическому листингу. Вот что говорит Мур:

Я структурирую иерархически `задачу`, а не обязательно `листинги`. Мои листинги организованы несколько неряшливо, не иерархически в том смысле, чтобы примитивы шли первыми.

Я использую слово LOCATE (известное также под именем VIEW; смотрите "Начальный курс...", стр.91). В результате листинг можно организовывать гораздо менее аккуратно, поскольку у меня есть LOCATE, которое мне все ищет. Я никогда не просматриваю листинги.

--> ПРОТИВ THRU.

При относительной загрузке одним из популярных способов загрузки серии смежных блоков является использование слова --> (читается "следующий блок"). Оно заставляет интерпретатор немедленно прекратить работу с текущим блоком и начать интерпретацию следующего (со следующим номером).

Если в Вашей системе есть -->, Вам придется делать выбор между использованием команды THRU в блоке загрузки главы и связыванием каждой последовательности воедино посредством стрелок и загрузкой с помощью LOAD лишь первого блока из серии.
(Не следует делать и то, и другое; все закончится загрузкой многих блоков более одного раза.)

Стрелочки хороши следующим: предположим, Вы меняете блок в середине серии, а затем перезагружаете его. Остальная часть серии также автоматически загружается. Вам не нужно помнить номер последнего нужного блока.

Есть у стрелок и недостатки: нет способа прервать процесс загрузки после его начала. Вам придется скомпилировать гораздо больше блоков, чем это нужно для тестирования Вашего единственного кусочка.

Если проанализировать все возможности, то обнаружатся три действия, которые Вам может понадобиться выполнить после внесения очередного изменения:

1. загрузить только один блок для проверки изменения,
2. загрузить всю секцию, в которую входит этот блок
   или
3. загрузить весь остаток задачи.

Использование слова THRU дает, кажется, наилучшие возможности управления.

Кое-кто считает, что стрелки полезны для того, чтобы позволить определению через двоеточие пересечь границу одного блока. Действительно, единственный путь для компиляции высокоуровневого определения (через двоеточие) изрядной длины - это использование слова -->, поскольку оно имеет признак "немедленного исполнения". Но НИКОГДА не может считаться хорошим стилем пересечение таким определнием границы блока. (Оно никогда не должно быть столь длинным!)

С другой стороны, чрезвычайно сложный и критичный ко времени исполнения кусок ассемблерного кода способен занимать несколько последовательных блоков. В этом случае все равно обычная загрузка вполне подходит, поскольку ассемблер не использует режим компиляции и поэтому не требует слов с признаком немедленного исполнения.

Наконец, стрелки требуют наличия дополнительной строки в каждом блоке. Мы не рекомендуем их применение.

АЛЬТЕРНАТИВА БЛОКАМ: ТЕКСТ В ИМЕНОВАННЫХ ФАЙЛАХ.

Некоторые пользователи Форта предпочитают хранение исходных текстов в именованных текстовых файлах переменной длины, намеренно имитируя подход, используемый в обычных компиляторах и редакторах. Такой путь может становиться все более и более распространенным, однако его полезность остается спорной.

Верно, хорошо не беспокоиться о том, что блока может не хватить, но ведь вынужденность использования ограниченного пространства блока компенсируется сохранением контроля над дискретными кусками кода. При разработке задачи Вы проводите гораздо больше времени, загружая и удаляя блоки, чем на переделывание их содержимого.

Файлы с "безграничной длиной" позволяют мыслить неорганизованно и беспорядочно и плохо разбивать задачу.

При отсутствии дисциплины, налагаемой границами 1К-байтного блока, определения становятся длиннее. Появляется тенденция писать 20-ти килобайтовые файлы или, что еще хуже, 20-ти килобайтовые определения.

Быть может, лучшим компромиссом будет основанная на файлах система, позволяющая вложенную загрузку и вдохновляющую на использование очень маленьких именованных файлов. При этом наиболее вероятно, что опытные Форт-программисты не будут использовать именованные файлы с длиной более, чем в 5-10К. Так в чем же выигрыш?

Кое-кто высказывает следующее утверждение: "легче запоминать имена, чем числа". Если это так, то достаточно лишь предопределить номера блоков как константы:

```
90 CONSTANT ОБРАМЛЕНИЕ
```

а затем для загрузки секции "обрамление" вводить

```
ОБРАМЛЕНИЕ LOAD
```

Или, для просмотра блока загрузки секции:

```
ОБРАМЛЕНИЕ LIST
```

(По соглашению, имена секций должны иметь окончания на "ИЕ" или аналогичные, соответствующие отглагольным существительным, в английском языке - на "ING" типа "framing".)

Конечно, для уменьшения ограничений подхода, основанного на использовании блоков, Вам нужны хорошие инструменты, включая команды редактора, перемещающие строки исходного текста из одного блока в другой и слова, передвигающие серии блоков вперед или назад внутри листинга.

РАЗБИЕНИЕ ДИСКА НА ЧАСТИ.

Последний аспект хорошо организованного листинга включает в себя стандартизацию соглашения, по которому определяется, что где происходит на диске. Такие стандарты должны вырабатываться на каждом предприятии или в отделе или индивидуальным программистом, в зависимости от сущности работы.

На рисунке 5-4 показана типичная для подразделения схема распределения места.

Рис.5-4. Пример схемы распределения дискового пространства внутри одного отдела.

Блок 0 - титульный блок, на нем показано наименование задачи, текущий номер версии и автор.

Блок 1 - блок загрузки задачи.

Блок 2 - зарезервирован для возможного продолжения блока 1.
Блоки 4 и 5 - содержат системные сообщения.

Блоки с 9 по 29 - утилиты общего назначения, нужные для отработки, но не для использования внутри задачи.

Блок 30 - начало блоков задачи.

На многих предприятиях, работающих с Фортом, считается важным начинать секции кода на блоках, номера которых нацело делятся на три. Основные разделы на диске должны иметь границы, проходящие по номерам, кратным тридцати.

Причина? По соглашению, блоки Форта распечатываются по три на страницу, причем начальный блок обычно имеет номер, кратный трем. Такая страница называется "триадой"; многие Форт-системы имеют слово TRIAD для формирования таких страниц, получая в качестве аргумента номер любой из трех страниц в триаде. К примеру, если набрать

```
77 TRIAD
```

то будет напечатана страница блоков с номерами 75, 76 и 77.

Основным преимуществом такого соглашения является то, что, если Вы исправляете один блок, то можете вложить новую триаду прямо в папку с распечаткой текущего листинга, заменяя при этом ровно одну страницу бумаги без перекрывающихся блоков.

Аналогично, слово INDEX показывает первые строчки из каждого блока, содержащегося в 60-ти блочной странице, если границы проведены на линиях, кратных 60-ти (*).

СОВЕТ
Начинайте секции или лексиконы на блоках, чьи номера кратны трем. Начинайте задачи или главы на номерах, кратных тридцати.

ВЫБОРКИ.

Изготовители Форт-систем сталкиваются с такой проблемой: если они включат в систему все команды, которые могут потребоваться покупателю - слова для графики, принтеров и другие штучки - то зачастую обнаруживается, что съедено более половины емкости памяти компьютера и осталось не так уж много места для того, чтобы серьезные программисты могли компилировать свои задачи.

Выходом для поставщика является разработка скелетного ядра с предварительно скомпилированными основными определениями, и отдельно - `исходных тестов`

расширений. Этот путь позволяет программисту выбрать и скомпоновать действительно нужные ему специальные программы.

(*) - это утверждение не соответствует описанию, приведенному в Приложении Б к стандарту Форт-83.

Такие загружаемые пользователем программы называются "выборками" (по-английски - "electives"). Арифметика двойной длины, поддержка печати даты и времени, конструкция CASE или DOER/MAKE (будут описаны позже) - это некоторые из тех вещей, которые Форт-система должна предлагать в качестве выборок.

ОФОРМЛЕНИЕ БЛОКА

В этом разделе мы обсудим правила оформления каждого из блоков с исходным текстом.

\---

СОВЕТ

Оставляйте строку 0 в качестве строки комментария.

\---

Строка комментария служит как для заголовка блока, так и для получения информации по диску словом INDEX. Она должна описывать назначение блока (но не содержать список определяемых в нем слов).

Как минимум, такая строка должна включать в себя наименование блока. В более крупных задачах можно также включить в нее и название главы. Если блок представляет собой часть серии блоков, реализующих лексикон, сюда же следует включить и "номер страницы".

Верхний правый угол резервируется для "штампа". В него входит дата последнего изменения и, когда это имеет значение, инициалы программиста (три буквы слева от даты), т.е.:

```
( Имя главы     Имя блока -- стр. #      АБВ 06/10/83 )
```

Некоторые Форт-редакторы сами проставляют штамп при нажатии специальной клавиши.

Обычной формой для представления даты (в Америке) является:

мм-дд-гг

то есть, февраль, 6-е число, 1984 года выражается как

02-06-84

Все возрастающую популярность приобретает такая альтернатива:

ддМммгг

где "Ммм" - это трехбуквенное сокращение месяца. К примеру:

22Окт84

Для нее требуется большее количество разных символов, чем для

10-22-84

и возникают возможности для перепутывания чисел и букв. (Для российских систем рекомендуется принятая в Европе и обычная у нас форма "дд-мм-гг" или, еще лучше, "дд.мм.гг".)

Если Ваша система имеет слово \ ("пропустить-строку" - см. приложение B), то можно писать строку комментария так:

```
\ Имя главы     Имя блока -- стр. #           АБВ 06/10/83
```

Как и во всех других примечаниях, используйте маленькие буквы или смесь из букв нижнего и верхнего регистров для текстов строки комментария.

Одним из путей достижения того, чтобы индекс приносил больше информации об организации файла - это установка отступа в строке комментария на три пробела для тех блоков, которые продолжают лексикон. На рисунке 5-5 показана порция распечатки, произведенной словом INDEX, в которой строки комментария для блоков-продолжений выполнены с отступом.

Рис.5-5. Результат работы слова INDEX, показывающий отступы в строках комментария.

```
90 \ Графика           Загрузка главы           АБВ 06/10/83
91    \ Примитивы рисования точки              АБВ 06/10/83
92 \ Примитивы рисования линий                 АБВ 06/10/83
93    \ Примитивы рисования линий              АБВ 06/10/83
94    \ Примитивы рисования линий              АБВ 06/11/83
95 \ Масштабирование, ротация                  АБВ 09/02/83
96    \ Масштабирование, ротация               АБВ 06/10/83
97    \ Масштабирование, ротация               АБВ 02/19/84
98    \ Масштабирование, ротация               АБВ 02/19/84
99 \ Прямоугольники                            АБВ 02/19/84
100 \ Круги                                    АБВ 06/10/83
101    \ Круги                                 АБВ 06/10/83
102    \ Круги                                 АБВ 06/10/83
```

СОВЕТ

Начинайте все определения с левого края блока, и определяйте не больше одного слова на строке.

`Плохо:`

```
: ПРИБЫТИЕ   ." ПРИВЕТ" ;    : ОТБЫТИЕ  ." ДО СВИДАНИЯ" ;
```

`Хорошо:`

```
: ПРИБЫТИЕ    ." ПРИВЕТ" ;
: ОТБЫТИЕ    ." ДО СВИДАНИЯ" ;
```

Это правило облегчает поиск определения в листинге. (Когда определения распространяются более, чем на одну строку, последующие строки всегда должны набираться с отступом.)

Переменные и константы (VARIABLE, CONSTANT) также следует определять по одной на строку. (См. "Примеры хорошего стиля комментирования" в приложении Д.) При этом остается место для объясняющего комментария в той же строке. Исключение составляют большие "семейства" слов (задаваемых специальным определяющим словом), для которых не требуются уникальные комментарии:

```
0 ОТТЕНОК ЧЕРНЫЙ       1 ОТТЕНОК СИНИЙ          2 ОТТЕНОК ЗЕЛЕНЫЙ
3 ОТТЕНОК ГОЛУБОЙ   4 ОТТЕНОК КРАСНЫЙ  5 ОТТЕНОК МАЛИНОВЫЙ
```

СОВЕТ

Оставляйте много места в конце блока для дальнейших добавлений.

При первичном написании программы заполняйте каждый блок кодом не более, чем на половину. Итеративный подход предполагает, что Вы вначале набрасываете компоненты задачи, а затем итеративно оживляете их до тех пор, пока все требования не окажутся выполненными. Обычно это означает добавление новых команд или поддержку особых случаев в существующих блоках. (Не `всегда`, однако. Новая итерация может привести к упрощению кода. Или добавленная сложность может в действительности относиться к другому компоненту и будет выделена в другой блок.)

Просторное размещение делает позднейшие добавления более приятными. Один писатель рекомендует при первом проходе заполнять блок кодом на 20-40 процентов и оставлять пустыми 80-60 процентов [1].

Не пропускайте строку между каждыми двумя определениями.
При этом все же можно пропустить строку между `группами` определений.

СОВЕТ

Все блоки должны оставлять систему счисления десятичной (DECIMAL).

Даже если у Вас встречаются подряд три блока, в которых используется шестнадцатеричная (HEX) система (к примеру, три блока на ассемблере) каждый из них должен начинаться с HEX на вершине и возвращаться к DECIMAL внизу. Это правило дает уверенность в том, что каждый блок может быть загружен отдельно для тестирования без привнесения грязи в положение дел. Кроме того, при чтении листинга Вы всегда знаете, что номера блоков - десятичные, независимо от потребностей самих блоков в системе счисления HEX.

На некоторых предприятиях это правило проводят даже дальше. Вместо того, чтобы по-простому исполнять DECIMAL в конце, на них возвращают основание системы счисления к `тому, каким оно было вначале`. Этот дополнительный предохранительный элемент может быть выполнен таким способом:

```
BASE @        HEX  \ сохранить исходн. состояние на стеке
0A2 CONSTANT ЗВОНКИ
```

```
0A4 CONSTANT СВИСТКИ
... и т.д. ...
BASE !          \ восстановить сост. системы счисления
```

Порой аргументы передаются через стек от блока к блоку, например, числа, возвращаемые словом BEGIN или IF в многоблочном ассемблерном определении, или базовый адрес, передаваемый от одного определяющего слова к другому - см. "Разбиение При Компиляции" в главе 6. В этих случаях лучше сохранять значение основания на стеке возвратов:

```
BASE @ >R      HEX
... и т.д. ...
R> BASE !
```

В некоторых организациях такой подход обязателен для любого блока, который меняет основание системы счисления, поэтому им не нужно об этом беспокоиться.

Мур предпочитает определять слово LOAD так, чтобы оно вызывало DECIMAL после загрузки. Такой подход упрощает содержимое блока, поскольку Вам самим не нужно заботиться о восстановлении основания.

ПРОПУСКИ И ОТСТУПЫ.

СОВЕТ

Без пропусков и отступов не может быть читаемости.

Примеры в этой книге соответствуют широко распространенным соглашениям по стилю пропусков и отступов. Читаемость обеспечивается правильно используемыми пустыми местами. При этом нет никакого проигрыша, кроме увеличенного расхода дешевой дисковой памяти.

Для тех, кто любит четко изложенные правила, в таблице 5-1 указаны основные постулаты. (Но помните, что для интерпретатора Форта нет ничего менее значительного, чем Ваши отступы и пропуски.)

Таблица 5-1. Основные отступы и пропуски.

----

1 пробел между : и именем
2 пробела между именем и его комментарием *
2 пробела или новая строка после комментария до тела определения *
3 пробела между именем и телом определения, если комментарии не используются
1 пробел между словами/числами внутри фразы
2 или 3 пробела между фразами
1 пробел между последним словом и ;
1 пробел между ; и IMMEDIATE (при необходимости)

Не ставить пустых строк между определениями, кроме случаев разграничения существенных групп определений.

* `Часто наблюдаемая альтернатива - 1 пробел между именем и комментарием и 3 - между комментарием и определением. Более либеральный подход использует по 3 пробела до и после комментария. Что бы Вы ни выбрали, твердо этого придерживайтесь.`

Последняя позиция в каждой строке должна быть пуста, кроме случаев, когда:

а) надписи в кавычках продолжаются на следующую строку, или
б) это - конец комментария.

Комментарий, начинающийся с \, может продолжаться до правого конца строки. Комментарии, начинающиеся с (, могут также иметь ограничительную) в последней позиции строки.

Вот некоторые частые ошибки при выполнении отступов и пропусков:

`Плохо` (имя не отделено от тела определения):

```
: ТОЛКАТЬ ВЗЯТЬСЯ НАЛЕЧЬ ;
```

`Хорошо:`

```
: ТОЛКАТЬ    ВЗЯТЬСЯ НАЛЕЧЬ ;
```

`Плохо` (последовательные строки без отступа в три пробела):

```
: СЛАВА  ( то-что-никогда-не-померкнет -- )
НЕ МЕРКНУТЬ НИКОГДА ;
```

`Хорошо:`

```
: СЛАВА  ( то-что-никогда-не-померкнет -- )
   НЕ МЕРКНУТЬ НИКОГДА ;
```

`Плохо` (нет разбиения на фразы):

```
: ГЕТТИСБУРГ   4 СЧЕТ 7 ЛЕТ + НАЗАД ;
```

`Хорошо:`

```
: ГЕТТИСБУРГ   4 СЧЕТ  7 ЛЕТ +  НАЗАД ;
```

Разбиение на фразы - искуство субъективное; я затрудняюсь предлагать какие-нибудь формальные правила.

СОГЛАШЕНИЯ ПО КОММЕНТАРИЯМ

Правильное составление комментариев обязательно. Имеются пять типов комментариев: комментарии по состоянию стека, по структуре данных, по входному потоку, по цели и повествовательные комментарии.

`Комментарий по стеку` показывает, какие аргументы определение берет со стека, а какие возвращает на стеке (если такие есть).

`Комментарий по структуре данных` показывает позицию и значение элементов этой структуры. К примеру, текстовый буфер может содержать счетчик в первом байте и 63 байта для текста.

`Комментарий по входному потоку` относится к тем строкам, которые слово собирается получить из входного потока. К примеру, слово Форта FORGET ищет имя словарного определения во входном потоке.

`Комментарий по цели определения` описывает, по возможности кратко, что делает данное определение. То, как оно работает, не должно заботить целевой компонент.

`Повествовательные комментарии` появляются внутри определения для объяснения того, что происходит, обычно строка за строкой. Такие комментарии используются исключительно в "вертикальном формате", который мы опишем в другом разделе.

Комментарии обычно записываются буквами нижнего регистра для отделения их от текста программы. (Большинство слов Форта записываются буквами верхнего регистра, маленькие буквы используют лишь в некоторых специальных случаях.)

В следующих разделах мы рассмотрим все стандартизованные форматы этих типов комментариев и дадим примеры на каждый из типов.

СТЕКОВАЯ НОТАЦИЯ.

СОВЕТ

Каждое определение через двоеточие или на ассемблере, которое снимает или кладет аргументы на стек, должно сопровождаться стековым комментарием.

"Стековая нотация" относится к соглашениям для представления того, что происходит на стеке. Формы такой нотации включают в себя "картинки стека", "изменения на стеке" и "комментарии изменений на стеке".

СТЕКОВАЯ КАРТИНКА.

Такая картинка изображает то, что предполагается находящимся на стеке в данный момент. Вещи перечисляются слева направо, причем слева находится дно стека, а справа – его вершина.

К примеру, стековая картинка

```
n1 n2
```

показывает лежащие на стеке два числа, причем наверху лежит n2 (в самой доступной позиции).

Это - тот же порядок, в котором Вы могли бы набрать эти числа, т.е. если n1=100, а n2=5000, можно было бы набрать

```
100 5000
```

для того, чтобы правильно положить их на стек.

Стековая картинка может содержать либо аббревиатуры типа "n1", либо полностью прописанные слова. Обычно используют первое. Некоторые из стандартных аббревиатур показаны в таблице 5-2. Независимо от того, используются ли аббревиатуры или полные слова, каждое из них должно быть отделено пробелом.

Если стековый комментарий описывается фразой (типа "адрес-последней-связи"), слова в такой фразе должны быть объединены черточками. К примеру, картинка

адрес текущий-отсчет макс-предел

показывает три элемента, находящихся на стеке.

ИЗМЕНЕНИЯ НА СТЕКЕ.

"Изменения на стеке" показывают две стековые картинки: первая изображает то, что определение может `потребить` со стека, а вторая - то, что оно на нем `возвращает`. Картинка "до" идет первой, после нее - два тире, а затем – картинка "после".

К примеру, вот стековые изменения для оператора сложения Форта - слова + :

n1 n2 -- сумма

Слово + берет два числа со стека и возвращает их сумму.

Помните, что изменения на стеке описывают лишь `чистый результат` выполняемой операции. Числа, которые, возможно, располагаются под используемыми аргументами, показывать не надо. Точно так же на надо показывать и те числа, которые появляются и исчезают на стеке во время исполнения определения. Если слово возвращает какие-нибудь входные аргументы неизмененными, то они должны быть повторены в выходной картинке:

3-й 2-й 1-входной -- 3-й 2-й 1-выходной

И наоборот, если слово изменяет какие-нибудь аргументы, то стековая картинка должна использовать другое изображение:

n1 -- n2
n -- n'

Изменения на стеке могут показываться и в сформатированном глоссарии.

КОММЕТАРИИ ИЗМЕНЕНИЙ НА СТЕКЕ.

"Комментарии изменений на стеке" - это описание таких изменений, появляющееся в исходном тексте в круглых скобках.
Вот стековый комментарий для слова COUNT:

(адрес-строки-со-счетчиком -- адрес-текста длина)

или:

('строки-со-счетчиком -- 'текста длина)

("Длина" после исполнения слова располагается на вершине стека.)

Если определение не оказывает влияния на стек (то есть, с точки зрения пользователя эффекта не наблюдается, независимо от того, насколько интенсивно используется стек внутри определения), то стековый комментарий не нужен:

```
: ПЕЧЬ    ЦЫПЛЯТА ПЕЧКА ! ;
```

С другой стороны, Вам может захотеться использовать пустой стековый комментарий - т.е.:

```
: ПЕЧЬ  ( -- )  ЦЫПЛЯТА ПЕЧКА ! ;
```

для подчеркивания отсутствия влияния слова на состояние стека.

Если определение берет аргументы, но ничего не возвращает, двойное тире необязательно, к примеру запись

```
( адрес длина -- )
```

может быть укорочена до

```
( адрес длина )
```

Такое соглашение принято на основании следующего наблюдения: гораздо чаще встречаются определения, которые берут аргументы и ничего не возвращают, чем те, которые ничего не берут, но возвращают на стеке результат.

СТАНДАРТНЫЕ АББРЕВИАТУРЫ ДЛЯ СТЕКОВЫХ КОММЕНТАРИЕВ.

Обозначения для стековой нотации должны быть содержательны. В таблице 5-2 показано большинство из наиболее часто используемых аббревиатур. (Эта таблица повторяется и в приложении Д.) Термины "одинарная длина", "двойная длина" и т.д. относятся к размеру "ячейки" данной Форт-системы. (Если система использует 16-разрядные машинные слова, то "n" представляет 16-битное число; если система работает с 32-мя разрядами, то "n" представляет 32-х разрядное число.)

ИЗОБРАЖЕНИЕ ФЛАГОВ.

В таблице 5-2 показаны три способа изображения булевских флагов. Для иллюстрации: вот три версии одного и того же стекового комментария для слова -TEXT:

```
( a1 u a2 -- ? )
( a1 u a2 -- t=не-равны )
( a1 u a2 -- f=равны )
```

Таблица 5-2. Обозначения для стековых комментариев.

----

n	чило одинарной длины со знаком
d	число двойной длины со знаком
u	чило одинарной длины без знака
ud	число двойной длины без знака
t	тройная длина
q	учетверенная длина
c	7 (или 8)-битный символ
b	8-ми битный байт
?	булевский флаг, или:
t=	(true) истина
f=	(false) ложь
a или adr или адр	адрес
acf	адрес поля кода
apf	адрес поля параметров
'	(в качестве префикса) адрес чего-либо
s d	(как пара) источник приемник
lo hi	нижняя- верхняя-граница (включительно)
#	число (количество)
o	(offset) смещение
i	индекс
m	маска
x	безразлично (для структур данных)

"Смещение" - это разница, выраженная в абсолютных единицах, например, байтах. "Индекс" - это разница, выраженная в логических единицах, например, элементах записи.

Знак равенства после символов "t" и "f" показывает, что этот флаг имеет определенное значение. Результат во втором варианте примера можно прочесть как "истина означает отсутствие равенства".

ЗАПИСЬ ДЛЯ РАЗЛИЧНЫХ ВАРИАНТОВ.

Некоторые определения при различных обстоятельствах по-разному воздействуют на стек.

Если количество чисел на стеке в любом случае остается неизменным, а меняются лишь функции этих чисел, можно использовать вертикальную черту (|) для обозначения "или".

Следующий комментарий изменения стека описывает слово, возвращающее либо адрес файла, либо нуль, если файл не найден:

(-- адрес|0=файла-нет)

Если количество чисел на стековой картинке может меняться – как на картинке "до", так и "после", - Вам следует писать обе версии полной стековой картинки, каждую - со своим двойным тире, разделенные символом "или". К примеру:

```
-FIND   ( -- apf len t=найдено | -- f=не-найдено )
```

Этот комментарий показывает, что если слово найдено, то на стеке возвращаются три аргумента (с флагом на вершине); иначе возвращается только флаг "ложь".

Обратите внимание на важность второго появления "--". Если его опустить, то это будет означать, что определение всегда возвращает три аргумента с флагом на вершине стека.

При желании можно записывать весь стековый комментарий дважды, либо в одной строке, отделяя записи тремя пробелами:

```
?DUP  \ если 0: ( n -- n)   если не-0: ( n -- n n )
```

либо по вертикали:

```
-FIND  \    найдено: ( -- apf len t )
       \ не-найдено: ( -- f )
```

КОММЕНТАРИИ К СТРУКТУРАМ ДАННЫХ.

"Комментарий к структуре данных" изображает элементы такой структуры. Для примера, вот определение буфера для вставки по имени |ВСТАВКА:

```
CREATE |ВСТАВКА  64 ALLOT  \   { 1# | 63текст }
```

Фигурные скобки начинают и заканчивают комментарий к структуре;
вертикальные черточки отделяют различные элементы в структуре;
числа представляют байты на элемент. В вышеприведенном комментарии первый байт содержит счетчик, остальные 63 отведены для текста.

"Битовый комментарий" использует такой же формат для отображения значения битов в байте или слове. К примеру, битовый комментарий

```
{ 1занят? | 1принят? | 2x | 6входное-устройство |
     6выходное-устройство }
```

описывает формат 16-ти битового регистра состояния коммуникационного канала. Первые два бита - флаговые, следующие два бита не используются и последняя пара 6-ти битных полей показывает, к какому входному и выходному устройству присоединен этот канал.

Если одну и ту же последовательность элементов использует более, чем одна структура данных, выпишите комментарий только один раз (возможно, в преамбуле), и дайте имя этой последовательности для последующих ссылок. К примеру, если в преамбуле вышеприведенной битовой последовательности дано имя "статус", то это имя можно употреблять в стековых комментариях для указания функции соответствующих чисел:

```
: СТАТУС?  ( -- статус)  ... ;
```

Если переменная двойной длины типа 2VARIABLE содержит одно число двойной длины, то комментарий должен быть стековой картинкой, показывающей это содержимое:

```
2VARIABLE ЦЕНА  \ цена в копейках
```

Если 2VARIABLE содержит два элемента одинарной длины, стековая картинка должна отображать то, что окажется на стеке после исполнения 2@. То есть:

```
2VARIABLE ИЗМЕРЕНИЯ  ( высота масса )
```

Это отличается от того, каким был бы комментарий, если бы ИЗМЕРЕНИЯ были бы определены через CREATE:

```
CREATE ИЗМЕРЕНИЯ  4 ALLOT   \ { 2масса | 2высота }
```

(Хотя в словаре оба определения будут представлены одинаково, использование 2VARIABLE предполагает, что значения будут обычно извлекаться и загружаться туда одновременно, с помощью 2! и 2@, и поэтому мы используем `стековый` комментарий. Число, появляющееся на вершине стека, указывается справа. При использовании CREATE имеется в ввиду, что эти значения будут извлекаться и загружаться по-отдельности - и мы используем комментарий для `структуры данных`. При этом 0-й элемент показывается слева.)

КОММЕНТАРИИ ДЛЯ ВХОДНОГО ПОТОКА.

Эти комментарии показывают, какие слова и/или строки предполагается извлекать из входного потока. В таблице 5-3 приведены сокращения, используемые для обозначения аргументов во входном потоке.

Таблица 5-3. Обозначения аргументов во входном потоке.

c	одиночный символ, выделенный пробелами
name или имя	последовательность символов, выделенная пробелами
text или текст	последовательность символов, выделенная не пробелами

После слова "текст" ставьте требуемый символ-ограничитель, типа: текст" или текст).

Комментарий входного потока приводится `до` стекового комментария и `не` заключается в собственную пару круглых скобок, а просто отделяется тремя пробелами с каждой стороны. К примеру, вот один способ комментирования определения слова ' (штрих), в котором вначале идет комментарий входного потока, а затем - комментарий стека:

```
: '  \  имя   ( -- a)
```

Если Вы предпочитаете использовать (, комментарий будет выглядеть так:

```
: '  (   имя   ( -- a)
```

Есть три различных способа получения входной строки. Для ясности обозначим соответствующие термины:

* `Сканирование` означает просмотр вперед по входному потоку, либо для получения слова или числа в случае ', либо для поиска ограничителя, как для слов ." или (.

* `Ожидание` означает использование слов EXPECT и KEY, а определения, которые их используют - "ожидают" ввода.

* `Предположение` показывает, что обычно что-то должно последовать. Слово : "сканирует" входной поток для получения имени определения и "предполагает", что последует тело этого определения.

Комментарий входного потока подходит только для случая использования сканирования.

ЦЕЛЕВЫЕ КОММЕНТАРИИ.

СОВЕТ
Каждое определение должно иметь целевой комментарий, кроме случаев, когда:
а) его работа ясна из стекового комментария, или
б) оно состоит из трех или меньшего числа слов.

Целевой комментарий должен иметь минимальные размеры – никогда не быть длиннее одной строки. К примеру:

```
: COLD   \ сбросить систему в исходное состояние
... ;
```

Используйте повелительное наклонение: "установить цвет фона", а не "устанавливает цвет фона".

С другой стороны, назначение слова может быть зачастую описано в терминах его стекового комментария. Нужда в обоих комментариях одновременно встречается не часто. Пример:

```
: SPACES  ( #)   ... ;
```

или

```
: SPACES  ( #пробелов-напечатать -- )   ... ;
```

Это определение получает в качестве входного аргумента число, представляющее количество пробелов, которые надо напечатать.

```
: ЭЛЕМЕНТ  ( #элемента -- 'элемента )   2*  ТАБЛИЦА + ;
```

Это определение преобразует получаемый им индекс в адрес внутри таблицы 2-х байтных элементов, относящийся к нужному элементу.

```
: PAD  ( -- 'временного-буфера)  HERE  80 + ;
```

Это определение дает адрес района памяти для временного использования.

Иногда читабельность лучше обеспечивается применением обоих типов комментариев. В этом случае целевой комментарий должен появляться последним. К примеру:

```
: BLOCK  ( n -- a)  \ закрепить блок n в буфере по адресу a
```

СОВЕТ
Показывайте тип комментария, соблюдая следующий порядок: вначале - комментарий по входному потоку, затем - комментарий изменений на стеке, последний - целевой комментарий.

К примеру:

```
: ДАТЬ   \   имя   ( -- a)   дать первое совпадение
```

Если Вы предпочитаете использовать (, можно записать:

```
: ДАТЬ   (   имя   ( -- a)   ( дать первое совпадение)
```

При необходимости можно расположить целевой комментарий в следующей строке:

```
: WORD   \   имя   ( c -- a)
   \ сканировать с символом-ограничителем c, получить по a
... ;
```

КОММЕНТАРИИ К ОПРЕДЕЛЯЮЩИМ СЛОВАМ.

При определении определяющего слова надо описывать два типа поведений:

определяющего слова, когда оно создает своего "потомка", и самого потомка (работу в режиме исполнения).

Эти два вида поведения следует комментировать по-отдельности.

СОВЕТ
Пишите комментарий к определяющему слову в режиме компиляции обычным образом; в режиме же исполнения отдельно, после слова DOES> (или ;CODE).

К примеру,

```
: CONSTANT  ( n )  CREATE ,
   DOES>  ( -- n )  @ ;
```

Комментарий изменений на стеке для поведения времени исполнения (т.е. потомка) показывает действие, производимое словом-потомком. При этом он не включает в себя изображение адреса, возвращаемого словом DOES>, хотя этот адрес и находится на стеке в начале исполнения потомка.

`Плохо` (комментарий времени исполнения содержит apf):

```
: МАССИВ   \   имя   ( #ячеек -- )
        CREATE  2* ALLOT
        DOES>   ( i apf -- 'ячейки)  SWAP 2* + ;
```

`Хорошо:`

```
: МАССИВ   \   имя   ( #ячеек -- )
        CREATE  2* ALLOT
        DOES>   ( i -- 'ячейки)  SWAP 2* + ;
```

Слова, определенные через МАССИВ, будут производить следующие действия на стеке:

```
( i -- 'ячейки)
```

Если определяющее слово не специфицирует поведения во время исполнения, это время все равно существует и может быть описано:

```
: VARIABLE  (   имя   ( -- )  CREATE  2 ALLOT ;
        \ does> ( -- адр)
```

КОММЕНТАРИИ ДЛЯ КОМПИЛИРУЮЩИХ СЛОВ.

Так же, как и для определяющих слов, большинство компилирующих слов дают два типа поведения:

1. Поведение компилирующего слова, когда оно появляется при компиляции очередного определения;
2. Поведение программы периода исполнения, когда вызвано слово, скомпилированное через данное компилирующее слово.

И вновь нам нужно писать комментарии раздельно для каждого из типов.

СОВЕТ

Пишите комментарий для поведения компилирующего слова в период исполнения обычным образом; комментируйте его поведение в режиме компиляции отдельно, начиная с пометки "Compile:" ("Компиляция:).

Пример:

```
: IF   ( ? -- )  ...
         \ Компиляция:  ( -- адр-неразрешенной-ссылки)
         ... ; IMMEDIATE
```

В компилирующих словах первая строка комментария описывает поведение в режиме исполнения, которое обычно и является `синтаксисом при использовании` слова. Второй комментарий описывает, что слово `в действительности делает` при компиляции (что менее важно для пользователя).

Еще примеры:

```
: ABORT"  ( ? -- )
         \ Компиляция:   текст"   ( -- )
```

Иногда компилирующее слово может по-другому вести себя при вызове `вне` определения через двоеточие. Такие слова (они вызывают брезгливость) требуют три строки комментария.
К примеру:

```
: ASCII  ( -- c)
         \ Компиляция:   c   ( -- )
         \ Интерпретация:   c   ( -- c )
         ... ; IMMEDIATE
```

В приложении Д приведены два блока, демонстрирующие хороший стиль комментирования.

ВЕРТИКАЛЬНЫЙ ФОРМАТ ЗАПИСИ ПРОТИВ ГОРИЗОНТАЛЬНОГО.

Целью комментирования является сделать читателю Вашего кода происходящее хорошо понятным. Однако сколько же нужно примечаний? Для понимания того уровня комментирования, который соответствует Вашим обстоятельствам, надо задать себе два вопроса:

Кто будет читать мой код?
Насколько удобочитаемы мои определения?

На выбор имеются два основных стиля комментирования. Первый, называемый "вертикальным форматом", содержит пошаговое описание процесса - так, как это делается в хорошо документированных ассемблерных программах. Такие построчные комментарии называются "повествовательными".

```
\ КЦК Контрольная сумма                    07/15/83
: НАКОПИТЬ   ( старКЦК символ -- новКЦК )
         256 *              \ сдвинуть символ в старший байт
         XOR                \ и искл.-ИЛИ со старым КЦК
         8 0 DO             \ затем восемь раз
             DUP 0< IF        \ если старший бит равен "1"
                 16386 XOR     \ искл.-ИЛИ с маской
```

```
        DUP +        \ и сдвинуть влево на 1 разряд
        1+           \ установить "1" в младшем бите
     ELSE      \ иначе, т.е. если старший бит - "0"
        DUP +        \ сдинуть влево на 1 разряд
     THEN      \
  LOOP ;          \ завершение цикла
```

При другом подходе кодовые фразы не перемежаются повествовательными комментариями. Он называется "горизонтальным форматом".

```
: НАКОПИТЬ   ( старКЦК символ -- новКЦК )
    256 *  XOR  8 0 DO  DUP 0< IF
    16386 XOR  DUP +  1+  ELSE  DUP +  THEN  LOOP ;
```

Вертикальный формат предпочтителен, когда над задачей работает большая команда программистов. Обычно такая группа включает в себя несколько программистов начального уровня, способных делать небольшие коррекции. При таком окружении по-строчное комментирование может сэкономить много времени и нервов. Как говорит Джонсон из Moore Products Co.: "При сопровождении программы обычно интересен один небольшой кусочек кода, и чем больше информации в нем написано, тем выше Ваши шансы быстро во всем разобраться".

Вот несколько уместных правил для Форт-программистов из Moore Products Co. (я перефразирую):

1. Должен использоваться вертикальный формат. Комментарии должны располагаться справа от исходного текста, но при необходимости могут занимать и всю следующую строку.
2. В комментариях должно быть больше символов, чем в относящимся к ним коде. (Фирма поощряет использование длинных описательных имен, более чем по десять символов длиной, и позволяет засчитывать имена в качестве комментариев.)
3. Любая структура управления или высокоуровневое слово должно появляться на отдельной строке. "Шумовые слова" могут группироваться вместе. Для показа вложенных условных переходов используются отступы.

Однако с таким форматом есть и некоторые сложности. С одной стороны, построчное комментирование отнимает много времени, даже при наличии хорошего экранного редактора. Продуктивность может резко упасть, особенно если остановки для написания комментариев прерывают ход Ваших мыслей.

Кроме того, приходится тщательно следить за тем, чтобы комментарии соответствовали действительности. Часто код меняется, версия проверяется, изменение работает – и программист при этом забывает поменять комментарий. Чем больше комментариев, тем больше вероятность того, что они неверны.
Если же они неверны, то они более чем бесполезны. Эта проблема может быть уменьшена, если руководитель проекта тщательно проверяет код и убеждается в точности примечаний.

Наконец, подробное комментирование может дать ложное чувство безопасности. Не надейтесь на то, что, поскольку каждая `строчка` имеет примечание, `задача` хорошо прокомментирована.
По-строчные примечания не подчеркивают важные аспекты работы определения. Что, к примеру, за мысль заложена в использованном алгоритме подсчета контрольной суммы? Как ее понять из повествовательных комментариев?

Для хорошего описания словами сущности данной процедуры требуется обычно много параграфов, а вовсе не одна фраза. Такие описания, скорее всего, должны быть представлены в иной документации или в преамбуле к главе.

Несмотря на все это, многие компании считают необходимым применение вертикального формата. Очевидно, что команда, начинающая работать с Фортом, должна его применять, так же, как и любая очень большая рабочая группа.

А как же горизонтальный формат? Быть может, это дело практики или личного искусства, но я просто обязан защитить горизонтальный формат как не менее ценный и, в некоторых случаях, более выигрышный.

Если Форт-код хорошо написан, то в нем не должно быть неясностей. Это означает, что:

* лексиконы поддержки имеют хорошо спроектированный синтаксис
* преобразования на стеке закомментированы
* выполнен целевой комментарий (если он не ясен из имени определения или стекового комментария)
* определения не слишком длинны
* не слишком много аргументов передается через стек одному определению (смотрите "Шикарный стек" в главе 7).

Форт просто не таков, как другие языки, в которых по-строчные примечания - это одна из немногих вещей, которые можно сделать для повышения читаемости программ.

Мастерски написанный на Форте код подобен поэзии, он содержит точное значение, которое и программист, и машина легко читают. Вашей `целью` должно быть написание такого кода, к которому примечания не нужны, даже если Вы захотите его комментировать. Проектируйте свои задачи так, чтобы код, а не примечания, нес в себе смысл.

Если Вы пойдете таким путем, то сможете устранить горы кропотливого комментирования, достигая чистоты выражения без избыточных объяснений.

СОВЕТ
Самой аккуратной и наименее дорогостоящей документацией
является самодокументированный код.

Рисунок в тексте:
Программист Уиггинс, гордящийся своей техникой комментирования.

```
Быстрая         ( установка для лисы; быстро движется)
коричневая      ( и имеет цвет шоколада)
лиса            ( исполнить лисицу)
прыгает через   ( дать действие лисицы)
ленивого        ( установка для собаки)
пса             ( то, через что лисица перепрыгнула)
```

К сожалению, даже лучшие программисты под давлением обстоятельств могут писать работающий код, который без примечаний читается нелегко. Если Вы пишете для себя, или работаете в маленькой группе, в которой есть взаимопонимание (на уровне слов),

горизонтальный формат идеален. В иных случаях придерживайтесь вертикального формата.

ВЫБОР ИМЕН: ИСКУССТВО

Кроме математических склонностей, исключительно хорошее владение своим родным языком - это одно из наиболее жизненно необходимых компетентному программисту качеств. (`Проф. Эдсжер У. Дийкстра [3]`).

Мы говорили о важности использования имен для обозначения идей или объектов в задаче. Выбор имен оказывается важной частью процесса проектирования.

Новички недооценивают вклад имен. "В конце концов," - думают они, - "компьютеру безразлично, какие наименования я выбираю".

Однако хорошие имена неотделимы от читабельности. Более того, мысленная попытка исполнения названного одним словом определения производит психологический эффект на Ваше представление о том, что должно, а что не должно делать целое.

Вот некоторые правила для выбора имен:

СОВЕТ
Выбирайте имена в соответствии с тем, `что`, а не с тем, `как`.

Определение должно скрывать от других определений сложности своей реализации. Кроме того, имя должно прятать детали процедур, описывая вместо них внешнее поведение или воздействие на стек.

К примеру, слово Форта ALLOT просто увеличивает указатель вершины словаря (называемый в большинстве систем DP или H). Однако имя ALLOT лучше имени DP+! тем, что пользователь думает о резервировании места, а не увеличении указателя.

В стандарте-83 принято имя CMOVE> вместо использовавшегося раньше функционально такого же <CMOVE. Эта операция позволяет перемещать район памяти `вперед` в перекрывающийся с ним район памяти. Достигается это тем, что копирование начинается с последнего байта и производится `назад`. В новом имени направление "вперед" свойства "что" главенствует над направлением "назад" свойства "как".

СОВЕТ
Ищите наиболее выразительные слова.

Верное слово - мощный помощник. Когда встречается нам такое совершенно точное слово в книге или газете, возникающее воздействие оказывается как физическим, так и духовным, дает эффект электрической подсказки (`Марк Твен`).

Различие между правильным словом и почти-правильным словом подобно различию между молнией и светящимся жучком (`Марк Твен`).

Suit the action to the word, the word to the action.
(Дай слову действенность, а действию дай слово.) (`Шекспир, "Гамлет", Акт III`).

Генри Лаксен, специалист по Форту и автор (Стандарта-83), считает, что самой важной частью инструментария Форта является хороший толковый словарь [4].

Иногда Вы колеблетесь над выбором слова для обозначения определения, чувствуя, что оно не совсем правильно. Могут пройти месяцы, прежде чем Вы осознаете, что промахнулись. В примере с римскими цифрами из главы 4 есть слово для обработки исключительного случая: чисел, которые на-единицу-меньше-чем значение следующего символа. Моим первым вариантом был 4-ИЛИ-9.
Это было ужасно, и только много позже я додумался до ПОЧТИ (по-английиски - ALMOST, в русском переводе менее осмысленно).

Большинство систем фиг-Форта содержат слово VLIST, которое распечатывает содержимое текущего словаря. Через много лет кто-то понял, что слово WORDS (СЛОВА) куда симпатичнее. Оно не только само по себе гораздо приятнее звучит, но также хорошо сочетается с именами словарей. К примеру:

```
EDITOR WORDS
```

или

```
ASSEMBLER WORDS
```

(правильные английские фразы). С другой стороны, Мур рассказывал, как неподходящие имена могут становиться средством криптозащиты. Если нужно обеспечить безопасность в тех случаях, когда Вас заставляют поставлять исходные тексты, код можно сделать чрезвычайно нечитаемым, сознательно выбирая уводящие от правильного смысла имена. Разумеется, при этом сопровождение становится невозможным.

СОВЕТ
Выбирайте такие имена, из которых можно составлять фразы.

Когда Вы не знаете, как бы назвать слово, подумайте о том, как оно будет использоваться в контексте. К примеру:

```
КРЫШКА ОТКРЫТЬ
```

ОТКРЫТЬ - это подходящее имя для слова, которое устанавливает бит в порту ввода/вывода по имени КРЫШКА.

```
3 КНОПКА ДЕЛАТЬ ЗАЖИГАНИЕ
```

ДЕЛАТЬ - хороший выбор для слова, которое выбирает функцию ЗАЖИГАНИЕ в таблице функций так, что ЗАЖИГАНИЕ будет исполнено при нажатии на кнопку 3.

```
СКАЗАТЬ ПРИВЕТ
```

СКАЗАТЬ правильно подобрано для установки вектора слова ПРИВЕТ на исполняемую переменную. (Когда я первый раз написал этот пример для "Способа мышления ...", то назвал это слово ВЕРСИЯ. Мур посмотрел рукопись и предложил СКАЗАТЬ, которое, несомненно, гораздо лучше.)

Я СЕРГЕЙ

Слово Я кажется более естественным, нежели LOGON СЕРГЕЙ, ПАРОЛЬ СЕРГЕЙ или СЕАНС СЕРГЕЙ, как это часто встречается.

Выбор слова Я (по-английски - I'M) - еще одно нововведение Мура, который по этому поводу говорит:

Мне не по вкусу слово LOGON. Нет такого слова в английском языке. Я хотел найти слово, с помощью которого говорят: "Я - ..." Это было так естественно. Я просто об него спотыкался.

Хоть оно и такое неуклюжее со своим апострофом (имеется в виду I'M), в нем присутствует ощущение правильности. Вот такие маленькие слова и составляют самый приятный путь к получению ощущения типа "Ага!".

Если раздумываете о выборе правильного слова - пусть оно будет `очевидно` верным. Чем шире Ваш активный запас слов, тем скорее Вы найдете подходящее.

Другое любимое слово Мура - это -Й (по-английски TH), которое используется для индексации внутри массива. Фраза вида

5 -Й

позволяет получить "пятый" элемент массива (его адрес).

СОВЕТ

Полностью пишите имена.

Однажды мне встретилась в одном журнале Форт-программа, автор которой, кажется, был одержим идеей исключать все гласные буквы из своих имен, порождая ужасы типа БФР-ДСПЛ вместо "буфера-дисплея". Другие же считают, что тремя буквами сказано все, и выделяют ДЛИ для "длины". Такая практика тормозит мышление.

Слова в Форте должны писаться полностью. Ощущайте гордость при набирании каждой буквы слова ИНИЦИАЛИЗИРОВАТЬ или ТЕРМИНАЛ или БУФЕР. Это - как раз те слова, которые Вы имеете в виду.

В сокращении слов еще хуже то, что постоянно забываешь, до какой степени их сократил. До ДСПЛ или до ДИСПЛ?

Другая проблема состоит в том, что такие сокращения мешают удобочитаемости. Любой язык программирования достаточно трудно читать и без того, чтобы примешивать еще и дополнительные сложности.

Однако и здесь есть некоторые исключения. Вот часть из них:

1. Слова, используемые в тексте чрезвычайно часто. В Форте есть пригоршня команд, которые употребляются вновь и вновь, не имея совсем или имея мало смысла:

: ; @ ! . ,

Но их такое ограниченное количество, и их применяют так часто, что они давно стали уже добрыми друзьями. Я бы никогда не захотел писать, на регулярной основе:

ОПРЕДЕЛИТЬ КОНЕЦ-ОПРЕДЕЛЕНИЯ ВЗЯТЬ ЗАГРУЗИТЬ
НАПЕЧАТАТЬ СКОМПИЛИРОВАТЬ#

(Интересно, что большинство этих символов не имеют синонимов в английском или русском языке. Мы говорим "определение `через двоеточие`", поскольку у нас нет другого термина; говорим "добавить `запятой` число к словарю" только потому, что это не совсем компиляция, а другого термина нет.)
2. Слова, которые оператор за терминалом будет часто использовать для управления работой. Такие имена хорошо определять однобуквенными, как это сделано для команд строкового редактора.
3. Слова, для которых обычно всегда используются аббревиатуры. Мнемоники в Форт-ассемблере обычно следуют тем, которые предлагаются производителями компьютеров, а они обычно бывают сокращенными (типа JMP или MOV).

Ваши имена должны быть произносимыми; в противном случае Вы раскаетесь, когда будете пытаться обсуждать программу с другими людьми. Если имя изображается символически, изобретите его произношение (к примеру, >R произносят "на-эр"; R> зовется "с-эр").

СОВЕТ
Любите короткие слова.

При выборе между трех-сложным и одно-сложным словом, означающим одно и то же, останавливайтесь на кратчайшем. ЯРКОСТЬ лучше, чем ИНТЕНСИВНОСТЬ. ВКЛЮЧИТЬ - более приемлемое имя, чем АКТИВИЗИРОВАТЬ; ПУСК, GO, RUN или ON могут оказаться еще лучше.

Короткие имена легче набирать. Они экономят место в блоке с исходным текстом. Что более важно, они делают код живым и чистым.

СОВЕТ
Чрезмерно усложненные имена могут быть признаком плохого качества разбиения.

Мур:

В Форт-среде встречаются разнообразные стили программирования. Некоторые используют сверхдлинные слова, выражающие по-английски свое назначение. Вы составляете вместе строчки из таких больших длинных слов и получаете что-то крайне хорошо читабельное.

Однако я тут же начинаю подозревать, что программист недостаточно хорошо продумал свои слова, что тире надо было бы разбить и составные части определить по-отдельности. Это не всегда возможно и не всегда приносит выгоду. Но я все равно подозреваю многочленное слово в смешивании двух концепций.

Сравните две следующие стратегии для одних и тех же вещей:

ВКЛЮЧИТЬ-ЛЕВЫЙ-МОТОР	ЛЕВЫЙ МОТОР ПУСК
ВКЛЮЧИТЬ-ПРАВЫЙ-МОТОР	ПРАВЫЙ МОТОР ПУСК
ВЫКЛЮЧИТЬ-ЛЕВЫЙ-МОТОР	ЛЕВЫЙ МОТОР СТОП
ВЫКЛЮЧИТЬ-ПРАВЫЙ-МОТОР	ПРАВЫЙ МОТОР СТОП
ВКЛЮЧИТЬ-ЛЕВЫЙ-МАГНИТ	ЛЕВЫЙ МАГНИТ ПУСК
ВКЛЮЧИТЬ-ПРАВЫЙ-МАГНИТ	ПРАВЫЙ МАГНИТ ПУСК
ВЫКЛЮЧИТЬ-ЛЕВЫЙ-МАГНИТ	ЛЕВЫЙ МАГНИТ СТОП
ВЫКЛЮЧИТЬ-ПРАВЫЙ-МАГНИТ	ПРАВЫЙ МАГНИТ СТОП

Синтакис левой части требует восьми словарных статей; синтаксис правой - только шести, и некоторые из этих слов, скорее всего, могут быть использованы и в других частях задачи. Если у Вас появятся "средние" мотор с магнитом, то число слов для описания всех 16-ти возможных ситуаций возрастет только до семи.

СОВЕТ
Не внедряйте числа в имена.

Следите за появлением имен, начинающихся или заканчивающихся цифрами, таких, как 1КАНАЛ, 2КАНАЛ, 3КАНАЛ и т.д.

Такое сращивание имен и чисел может указывать на плохо выполненное разбиение. Здесь состав преступления тот же, что и в сверхдлинных словах, только отделить нужно было не слово, а число. Лучше было бы разбить вышеуказанное так:

```
1 КАНАЛ
2 КАНАЛ
3 КАНАЛ
```

В этом случае необходимые три слова сократились до одного.

Зачастую скрещивание чисел со словами может означать недостаток фантазии при выборе имен. В вышеприведенном случае более описательными могли бы оказаться, например, имена

```
МИКРОФОН ТЕЛЕМЕТРИЯ ГИТАРА
```

Мы еще дадим развитие всем этим соображениям в следующей главе при "фрагментации".

СТАНДАРТЫ ПРИ ВЫБОРЕ ИМЕН: НАУКА

СОВЕТ
Изучите и используйте соглашения по выбору имен в Форте.

Для создания коротких, но наполненных значением слов Форт-программисты приняли некоторые соглашения. В приложении Д приводится наработанный за многие годы список из таких наиболее полезных соглашений.

Примером мощи соглашений может служить использование "точки" для обозначения "печати" или "вывода". Сам Форт имеет слова

```
. D. U.R
```

для вывода различных типов чисел в разных форматах. Соглашения также распространяются и на целевые слова. Если у Вас имеется переменная по имени ДАТА, и Вам нужно слово, которое печатает эту дату, используйте имя

```
.ДАТА
```

Замечание: чрезмерное увлечение префиксами и суффиксами делает слова уродливыми и совсем нечитаемыми. Не пытайтесь описать все, что делает слово только в его имени. В конце концов, имя - это просто символ, а не краткое содержание кода. Что звучит лучше и лучше читается?:

Эдипов комплекс

(что само по себе ничего не означает), или

подсознательное-влечение-к-родителю-иного-пола комплекс

Наверное, первое, хотя оно и предполагает, что Вы в курсе дела.

СОВЕТ
Используйте префиксы и суффиксы для подчеркивания отличий
между похожими друг на друга словами вместо того, чтобы
с их помощью расписывать значения слов внутри имен.

К примеру, фраза

```
... ГОТОВО IF ЗАКРЫТЬ THEN ...
```

так же читаема, как и

```
... ГОТОВО? IF ЗАКРЫТЬ THEN ...
```

и при этом выглядит более чистой. Поэтому она предпочтительна, если только нам не требуется дополнительное слово ГОТОВО (например, в качестве флага).

Последний совет по именам:

СОВЕТ
Начинайте все шестнадцатеричные числа с "0" (нуль) для
исключения потенциальной путаницы с именами.

Например, пишите 0ADD, а не ADD.

Между прочим, не очень рассчитывайте на то, что Ваша Форт-система поддерживает все эти соглашения. Предполагается, что такие соглашения должны применяться для новых разработок.

Форт был создан и многие годы улучшался людьми, которые использовали его в хвост и в гриву. В те времена было не нужно, да и невозможно воздвигать стандарты на имена для инструмента, который все еще рос и углублялся.

Если бы Форт был спроектирован каким-нибудь комитетом, мы бы его так не любили.

ЕЩЕ СОВЕТЫ ПО ЧИТАБЕЛЬНОСТИ

Вот несколько заключительных предложений для повышения удобочитаемости Вашего кода. (Определения приводятся в приложении В).

Всегда окупающей себя в большинстве задач константой является BL (код ASCII для "пробела").

Слово ASCII применяется, в основном, внутри определений через двоеточие для того, чтобы освобождать нас от необходимости знания численного значения символов ASCII. К примеру, вместо того, чтобы писать

```
: (   41 WORD  DROP ; IMMEDIATE
```

помня, что 41 - это код ASCII для закрывающей скобки, можно написать

```
: (   ASCII ) WORD  DROP ; IMMEDIATE
```

Сделать работу с булевскими значениями более наглядной могут слова TRUE и FALSE (ИСТИНА и ЛОЖЬ). Эти добавления позволят Вам писать выражения типа

```
TRUE 'МАРКА? !
```

для установки флага, или

```
FALSE 'МАРКА? !
```

для его очистки.

(Когда-то я использовал T и F, но они мне нужны так редко, что теперь я осмотрительно придерживаюсь соглашения против сокращений.)

Частью Вашей задачи (не обязательно частью Вашей Форт-системы) может стать следующий шаг в развитии этой идеи:

```
: ON  ( a)  TRUE SWAP ! ;
: OFF  ( a)   FALSE SWAP ! ;
```

Эти слова позволят Вам писать:

```
'МАРКА? ON
```

или

```
'МАРКА? OFF
```

Эти определения встречаются и под другими именами, типа SET и RESET (УСТАНОВИТЬ и СБРОСИТЬ), хотя обычно слова с такими именами используют маски для манипуляций отдельными битами.

Слово WITHIN (МЕЖДУ) также используется часто. Оно определяет, находится ли данное значение в интервале между двумя другими числами. Синтаксис его таков:

```
n  lo hi  WITHIN
```

Число "n" подлежит проверке, а "lo" и "hi" представляют диапазон. Слово WITHIN возвращает истину (TRUE), если "n" `больше или равно` "lo" и `меньше` "hi". Такое использование верхнего ограничителя (его исключение) соответствует аналогичному синтаксису циклов DO LOOP.

Мур рекомендует использовать слово UNDER+ (ПОД+). Оно полезно для складывания между собой чисел не на вершине стека, а под его вершиной. На высоком уровне его можно было бы
определить так:

```
: UNDER+  ( a b c -- a+b c )  ROT +  SWAP ;
```

ИТОГИ

Поддерживаемость задачи требует ее удобочитаемости. В этой главе мы перечислили много путей для того, чтобы делать код более читабельным. Мы одобрили курс на написание сколь это только возможно более самодокументированных программ. Технические приемы при этом касаются организации листингов, отступов и пробелов в тексте, комментирования, выбора имен и дополнительных слов, повышающих ясность кода.

Мы только вскользь упомянули внешнюю документацию, которая представляет собой все, что не входит в собственно листинг. Мы не будем ее в дальнейшем обсуждать, однако она все равно остается неотъемлемой частью процесса создания программного обеспечения.

ЛИТЕРАТУРА

1. Gregory Stevenson, "Documentation Priorities," `1981 FORML Conference Proceedings`, p. 401.
2. Joanne Lee, "Quality Assurance in a FORTH Environment," (Appendix A), `1981 FORML Proceedings,` p.363.
3. Edsger W. Dijkstra, `Selected Writings on Computing: A Personal Perspective,` New York, Springer Verlag, Inc., 1982.
4. Henry Laxen, "Choosing Names," `FORTH Dimensions,` vol. 4, no. 4, FORTH Interest Group.

ГЛАВА 6.

ФРАГМЕНТАЦИЯ

В этой главе мы продолжим наше исследование фазы реализации, сосредоточившись на этот раз на фрагментации.

Декомпозиция и фрагментация - суть разные части единого целого. И то, и другое включает в себя разбиение и организацию. Декомпозиция применяется при предварительном проектировании, в то время, как фрагментация - при детализированной проработке и реализации.

Поскольку каждое определение через двоеточие отражает то, какие были приняты решения при фрагментации, владение хорошей техникой этого процесса является едва ли не самым важным умением для Форт-программиста.

Так что же это такое? Фрагментация означает организацию кода в полезные куски. Для того, чтобы они оказались полезными, часто необходимо отделить фрагменты, которые могут быть использованы повторно, от тех, которые встречаются однократно. Первые становятся новыми определениями. Вторые преобразуются в аргументы или параметры для определений.

Такое разделение обычно называют "факторизацией". Первая часть это главы будет посвящена обсуждению различных приемов для такой "факторизации".

Другой стороной фрагментации является принятие решений о том, сколь много надо оставить внутри, а сколько вынести наружу определения. Во второй части будут приведены критерии для разумной и полезной фрагментации.

ТЕХНИКА ФАКТОРИЗАЦИИ

Если модуль кажется почти, но не совсем полезным с точки зрения еще какого-нибудь места в системе, попробуйте найти и выделить полезную подфункцию. Оставшаяся часть модуля может быть перенесена в вызывающую часть (из `Структурированного проектирования` [1]).

Разумеется, "полезная подфункция" становится вновь вычлененным определением. А что такое "не совсем полезная"? Это зависит от того, что она собой представляет.

ФАКТОРИЗАЦИЯ ДАННЫХ.

Проще всего выделить данные, за что следует благодарить Фортов стек данных. К примеру, для вычисления двух третей от 1000, мы пишем

```
1000 2 3 */
```

Для определения слова, вычисляющего две трети от `любого` числа, мы отделяем аргумент от определения:

```
: ДВЕ-ТРЕТИ ( n1 -- n2 ) 2 3 /* ;
```

Когда данные должны применяться в `середине` полезного выражения, нам приходится использовать манипуляции со стеком. К примеру, для того, чтобы расположить по центру 80-знаковой строки экрана текст длиной в 10 символов, можно написать:

```
80 10 - 2/ SPACES
```

Однако текст не всегда бывает длиной по 10 символов. Для придания полезности такой фразе при любой строке, следовало бы отделить длину, написав:

```
: ЦЕНТРОВАТЬ ( длина -- ) 80 SWAP - 2/ SPACES ;
```

Стек данных может также использоваться и для передачи адресов. Поэтому то, что выделено, может быть `указателем` на данные, вместо самих данных. Данные же могут быть числами и даже строками, но все равно будут таким образом отделены благодаря использованию стека.

Иногда различие оказывается функцией, которая, однако, может быть легко приведена к числу, передаваемому через стек. К примеру:

```
Сегмент 1: ВИЛЛИ НИЛЛИ ПУДИНГ ПИРОГ AND
Сегмент 2: ВИЛЛИ НИЛЛИ 8 * ПУДИНГ ПИРОГ AND
```

Как можно факторизовать операцию "8 *"? Оставляя "*" во фрагменте, и передавая ему единицу или восьмерку:

```
: НОВАЯ ( n -- ) ВИЛЛИ НИЛЛИ * ПУДИНГ ПИРОГ AND ;
Сегмент 1: 1 НОВАЯ
Сегмент 2: 8 НОВАЯ
```

(Конечно, если ВИЛЛИ или НИЛЛИ меняют состояние стека, Вам понадобится добавить подходящие стековые операторы.)

Если операция производит сложение, то ее можно обойти, передавая фрагменту ноль.

СОВЕТ
Для простоты пытайтесь представить различия в похожих фрагментах как числовые (по значениям или адресам), вместо того, чтобы представлять их как процедурные.

ВЫДЕЛЕНИЕ ФУНКЦИЙ.

С другой стороны, различие иногда представляется `только` функцией. Дано:

```
Сегмент 1: ВЗДОР-А ВЗДОР-Б ВЗДОР-В
           ВЗДОР-Г ВЗДОР-Д ВЗДОР-Е
Сегмент 2: ВЗДОР-А ВЗДОР-Б ПЕРЕВОРОТ
           ВЗДОР-Г ВЗДОР-Д ВЗДОР-Е
```

Неправильный подход:

```
: ВЗДОРЫ  ( t=делать-ВЗДОР-В | f=делать-ПЕРЕВОРОТ -- )
        ВЗДОР-А ВЗДОР-Б  IF  ВЗДОР-В  ELSE  ПЕРЕВОРОТ  THEN
        ВЗДОР-Г ВЗДОР-Д ВЗДОР-Е ;

Сегмент 1:   TRUE ВЗДОРЫ
Сегмент 2:   FALSE ВЗДОРЫ
```

Более подходящий вариант:

```
: ВЗДОР-АБ    ВЗДОР-А  ВЗДОР-Б ;
: ВЗДОР-ГДЕ    ВЗДОР-Г  ВЗДОР-Д  ВЗДОР-Е ;

Сегмент 1:   ВЗДОР-АБ  ВЗДОР-В  ВЗДОР-ГДЕ
Сегмент 2:   ВЗДОР-АБ  ПЕРЕВОРОТ  ВЗДОР-ГДЕ
```

СОВЕТ
Не передавайте управляющих флагов нижестоящим словам.

А почему нет? Во-первых, Вы требуете от Вашей исполняемой задачи принятия ненужного решения - того, ответ на который Вам и так ясен при программировании - и этим снижаете эффективность.

Во-вторых, терминология не соответствует концептуальной модели. Что означают TRUE ВЗДОРЫ (правильные вздоры) в противоположность FALSE ВЗДОРЫ (неправильным вздорам)?

ФАКТОРИЗАЦИЯ КОДА ИЗ СТРУКТУР УПРАВЛЕНИЯ.

Остерегайтесь повторений в обеих ветвях выражений типа IF THEN ELSE. К примеру:

```
... ( c)  DUP  BL  127 WITHIN
          IF  EMIT  ELSE
          DROP  ASCII . EMIT    THEN ...
```

Этот фрагмент печатает ASCII-символ, кроме тех случаев, когда этот символ - управляющий, в этом случае печатается точка. В любом случае выполняется слово EMIT. Следует выделить EMIT из структуры управления, например:

```
... ( c)  DUP  BL  127 WITHIN
          IF  DROP  ASCII .   THEN   EMIT ...
```

Хуже всего обстоит дело, когда различие между двумя определениями проявляется как функция внутри структуры, что делает выделение частей фрагмента невозможным. В такой ситуации используйте стековые аргументы, переменные или даже векторизацию. Как можно использовать векторизацию, мы покажем в разделе главы 7, названном "Использование DOER/MAKE".

Вот напоминание на случай факторизации кода из циклов DO LOOP:

СОВЕТ
При факторизации содержимого циклов DO LOOP в отдельное определение переработайте код таким образом, чтобы слово I (индекс) не употреблялось внутри такого определения, но передавалось ему через стек.

ФАКТОРИЗАЦИЯ САМИХ СТРУКТУР УПРАВЛЕНИЯ.

Вот два определения, отличающиеся внутренностью конструкции IF THEN:

```
: АКТИВНЫЙ
        А Б OR  В AND  IF  БЕСИТЬСЯ РЕЗВИТЬСЯ ПРЫГАТЬ  THEN ;

: ЛЕНИВЫЙ
        А Б OR  В AND  IF   СИДЕТЬ   ЕСТЬ   СПАТЬ     THEN ;
```

Условие и структура управления не отличаются; лишь события другие. Поскольку выделить IF в одно слово, а THEN в другое нельзя, проще всего факторизовать условие:

```
: УСЛОВИЯ?  ( -- ?)  А Б OR  В AND ;
: АКТИВНЫЙ
        УСЛОВИЯ?  IF  БЕСИТЬСЯ РЕЗВИТЬСЯ ПРЫГАТЬ  THEN ;
: ЛЕНИВЫЙ
        УСЛОВИЯ?  IF   СИДЕТЬ   ЕСТЬ   СПАТЬ     THEN ;
```

При большом количестве повторений одного и того же условия и структуры управления можно даже выделить и то, и другое. Смотрите:

```
: УСЛОВНО   А Б OR  В AND NOT  IF  R> DROP  THEN ;
: АКТИВНЫЙ   УСЛОВНО  БЕСИТЬСЯ РЕЗВИТЬСЯ ПРЫГАТЬ ;
: ЛЕНИВЫЙ    УСЛОВНО  СИДЕТЬ   ЕСТЬ   СПАТЬ ;
```

Слово УСЛОВНО может - в зависимости от условия - изменять поток управления таким образом, что остальные слова в каждом определении исполнены не будут. У такого подхода есть и определенные недостатки. Мы будем обсуждать такую технику – за и против - в 8-й главе.

Менее жестокие примеры факторизации структур управления основываются на выражениях типа CASE, устраняющие вложенные IF THEN ELSE, и множественные выходы из циклов (конструкции BEGIN WHILE WHILE WHILE ... REPEAT). Мы еще обсудим эти темы в главе 8.

ФАКТОРИЗАЦИЯ ИМЕН.

Хорошо также факторизовывать имена в случаях, когда они кажутся почти, но не совсем одинаковыми. Просмотрите следующий ужасающий пример кода, которые предназначен для инициализации трех переменных для каждого из восьми каналов:

```
VARIABLE 0STS       VARIABLE 1STS       VARIABLE 2STS
VARIABLE 3STS       VARIABLE 4STS       VARIABLE 5STS
VARIABLE 6STS       VARIABLE 7STS       VARIABLE 0TNR
VARIABLE 1TNR       VARIABLE 2TNR       VARIABLE 3TNR
VARIABLE 4TNR       VARIABLE 5TNR       VARIABLE 6TNR
VARIABLE 7TNR       VARIABLE 0UPS       VARIABLE 1UPS
VARIABLE 2UPS       VARIABLE 3UPS       VARIABLE 4UPS
VARIABLE 5UPS       VARIABLE 6UPS       VARIABLE 7UPS

: INIT-CH0   0 0STS !   1000 0TNR !   -1 0UPS ! ;
: INIT-CH1   0 1STS !   1000 1TNR !   -1 1UPS ! ;
: INIT-CH2   0 2STS !   1000 2TNR !   -1 2UPS ! ;
: INIT-CH3   0 3STS !   1000 3TNR !   -1 3UPS ! ;
: INIT-CH4   0 4STS !   1000 4TNR !   -1 4UPS ! ;
: INIT-CH5   0 5STS !   1000 5TNR !   -1 5UPS ! ;
: INIT-CH6   0 6STS !   1000 6TNR !   -1 6UPS ! ;
: INIT-CH7   0 7STS !   1000 7TNR !   -1 7UPS ! ;

: INIT-ALL-CHS   INIT-CH0  INIT-CH1  INIT-CH2  INIT-CH3
      INIT-CH4  INIT-CH5  INIT-CH6  INIT-CH7 ;
```

Во-первых, имеется сходство между именами переменных, кроме того, сходство есть и в коде, используемом в словах INIT-CH.

Вот улучшенный вариант. Одинаковые имена переменных были факторизованы в три структуры данных, а длинные процедуры в словах INIT-CH были вынесены в структуру DO LOOP:

```
: МАССИВ  ( #ячеек -- )  CREATE  2* ALLOT
      DOES> ( i -- 'ячейки)  SWAP 2* + ;
8 МАССИВ STATUS   ( #канала -- адр )
8 МАССИВ TENOR    (       "       )
8 МАССИВ UPSHOT   (       "       )
: УСТАНОВИТЬ   8 0 DO   0 I STATUS !  1000 I TENOR !
-1 I UPSHOT !  LOOP ;
```

Вот весь нужный нам код.

Даже и в более невинных случаях создание маленькой структуры данных может сократить количество дополнительных имен. Имеется соглашение, по которому в Форте текст хранится в "строках со счетчиком" (т.е. с количеством символов, хранимом в первом байте). Любое слово, возвращающее "адрес строки", на самом деле дает начальный адрес, адрес счетного байта. Такая двухэлементная структура данных не только убирает необходимость в раздельных именах для строки и ее длины, но также облегчает перемещение такой строки в памяти, поскольку и строку, `и` ее счетчик можно скопировать сразу, одним словом CMOVE.

Когда там и здесь Вам начинают попадаться разные ужасы, что-то можно скомбинировать, да и убрать их с глаз долой.

ФАКТОРИЗАЦИЯ ФУНКЦИЙ В ОПРЕДЕЛЯЮЩИЕ СЛОВА.

СОВЕТ
Если последовательности определений содержат одинаковые функции, отличающиеся лишь подставляемыми данными, используйте определяющее слово.

Рассмотрите структуру этого кода (не беспокоясь об его назначении - позже такой пример Вам еще встретится):

```
: ОТТЕНОК  ( цвет -- цвет' )
    'СВЕТЛЫЙ? @  OR   0 'СВЕТЛЫЙ? ! ;
: ЧЕРНЫЙ   0 ОТТЕНОК ;
: СИНИЙ    1 ОТТЕНОК ;
: ЗЕЛЕНЫЙ  2 ОТТЕНОК ;
: ЖЕЛТЫЙ   3 ОТТЕНОК ;
: КРАСНЫЙ  4 ОТТЕНОК ;
: ФИОЛЕТОВЫЙ  5 ОТТЕНОК ;
: КОРИЧНЕВЫЙ  6 ОТТЕНОК ;
: СЕРЫЙ    7 ОТТЕНОК ;
```

Такой подход корректен, однако менее эффективен с точки зрения занимаемой памяти, чем следующий, в котором используется определяющее слово:

```
: ОТТЕНОК  ( цвет -- )  CREATE ,
    DOES>  ( -- цвет)  @  'СВЕТЛЫЙ? @  OR   0 'СВЕТЛЫЙ? ! ;
0 ОТТЕНОК ЧЕРНЫЙ  1 ОТТЕНОК СИНИЙ      2 ОТТЕНОК ЗЕЛЕНЫЙ
3 ОТТЕНОК ЖЕЛТЫЙ  4 ОТТЕНОК КРАСНЫЙ   5 ОТТЕНОК ФИОЛЕТОВЫЙ
6 ОТТЕНОК КОРИЧНЕВЫЙ     7 ОТТЕНОК СЕРЫЙ
```

(Суть определяющих слов объясняется в книге "Начальный курс программирования ...", в главе 11).

Используя определяющее слово, мы экономим память, поскольку каждому двоеточечному определению необходим адрес слова EXIT для выхода. (При определении восьми слов использование определяющего слова экономит 14 байтов для 16-ти битового Форта.) Кроме того, в определениях через двоеточие каждая ссылка на числовой литерал требует компиляции ссылки на слово LIT (или literal), то есть еще по 2 байта на определение.

Если числа 1 и 2 - это предварительно определенные константы, то за это приходится расплачиваться еще 10-ю байтами – итого 24 байта.)

С точки зрения читабельности определяющее слово делает абсолютно ясным то обстоятельство, что все вводимые с его помощью цвета принадлежат к одному семейству слов.

Однако самая большая сила определяющих слов проявляется тогда, когда множество определений разделяют одно и то же поведение `во время компиляции`. Эта тема будет предметом для обсуждения в последующем разделе, "Факторизация во время компиляции".

КРИТЕРИИ ДЛЯ ФРАГМЕНТАЦИИ

Теперь, вооружившись техникой факторизации, давайте обсудим некоторые критерии разбиения определений в Форте. Вот они:

1. Ограничение размера определения
2. Ограничение повторов в коде
3. Оптимизация имен
4. Упрятывание информации
5. Упрощение командного интерфейса

СОВЕТ
Пусть определения будут короткими.

Мы задали Муру вопрос: "Какой длины должно быть определение на Форте?"

> Слово должно быть длиной в одну строку. Это цель. Когда у Вас имеется много слов, каждое из которых в своем роде полезно - быть может, лишь для отладки или апробирования, но причина для его существования обязательно есть - тогда Вы ощущаете, что ухватили самую сущность проблемы и это - те слова, которые ее выражают. Короткие слова дают хорошее качество такого ощущения.

Беглый осмотр одной из написанных Муром программ показал, что в среднем на определение у него приходится по семь ссылок, в том числе и на числа, и на слова. Эти определения замечательно коротки. (Реально его код 50 на 50 состоит из одно- и двухстрочных определений.)

Психологические тесты показали, что человеческое сознание может сосредоточиться только на семи, плюс-минус двух, вещах одновременно [2]. В то же время постоянно, днем и ночью, огромные ресурсы нашего ума заняты подсознательным сбором непрерывных потоков данных, наведением связей и группированием и решением задач.

Даже если наше подсознание и знает вдоль и поперек все уголки задачи, наше ограниченное сознание может одновременно сочетать лишь семь вещей сразу. За этими пределами наша хватка теряет прочность. Короткие определения соответствуют возможностям нашего разума.

Многих Форт-программистов побуждает писать слишком длинные определения знание того, что заголовки занимают место в словаре. Чем крупнее разбиение, тем меньше имен и тем меньше памяти будет потрачено.

Это правда, память будет использована, но вряд ли можно сказать, что нечто, помогающее Вам проверять, отлаживать и взаимодействовать со своим кодом - это "трата". Если у Вас большая задача, попытайтесь использовать по умолчанию ширину поля имени, равную трем, с возможностью переключения на полную длину во избежание

коллизий. (Ширина - WIDTH – содержит предельное число символов, которое может храниться в поле имени каждого из заголовков в словаре.)

Если задача все еще остается слишком большой, используйте Форт со множественными словарями на машине с расширенной памятью или, еще лучше, 32-х битный Форт на ЭВМ с 32-разрядной адресацией.

Родственный страх возникает и от того, что чрезмерное дробление будет снижать производительность за счет слишком частого обращения к встроенному интерпретатору Форта.

Опять-таки, правда то, что за каждый уровень вложенности приходится платить. Но обычно проигрыш за вложенность при правильном разбиении не заметен. Если же положение действительно так серьезно, правильным решением будет перевод некоторых частей на ассемблер.

СОВЕТ
Производите разбиение там, где чувствуете неуверенность в своем коде (где сложность достигает пределов понимания).

Не позволяйте своему Я бравировать отношением "Я это превзойду!". Код на Форте никогда не должен вызывать чувство неуютной сложности. Факторизуйте!

Мур:

Ощущение того, что дело идет к ошибке, является одним из поводов к расчленению. Всякий раз, встречая цикл DO LOOP двойной вложенности, Вы должны видеть указание на то, что что-то неверно, поскольку это будет трудно отлаживать.

Почти всегда хорошо взять внутренний DO LOOP и сделать его словом. И, выделив слово для отладки, нет никакой причины внедрять его назад. Оно пригодилось вам вначале. Нет гарантии того, что оно опять Вам не понадобится.

Вот еще одна сторона того же принципа:

СОВЕТ
Производите факторизацию в той точке, где кажется необходимым применение комментария.

В частности, если Вы ощущаете, что нужно напомнить себе, что лежит на стеке, то это может оказаться тем самым моментом, когда хорошо "сделать перерыв".

Предположим, у Вас написано:

```
... БАЛАНС  DUP xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx
        xxx xxx xxx xxx xxx xxx xxx  ( баланс) ПОКАЗАТЬ  ...
```

Ситуация начинается с вычисления баланса и заканчивается его распечаткой. Посередине же несколько строчек кода используют баланс для собственных нужд. Поскольку трудно сразу увидеть, что баланс все еще находится на стеке при вызове ПОКАЗАТЬ, программист вставляет стековую картинку.

Такое решение обычно говорит о плохой факторизации. Лучше написать:

```
: РЕВИЗОВАТЬ  ( баланс -- )  xxx xxx xxx xxx xxx xxx xxx
        xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx xxx ;
... БАЛАНС  DUP РЕВИЗОВАТЬ  ПОКАЗАТЬ  ...
```

При этом не нужны внутренние стековые картинки. Более того, программист теперь получил пригодный для повторного использования, проверяемый набор определений.

СОВЕТ

Ограничивайте повторения кода.

Другой причиной для факторизации является устранение повторяющихся фрагментов кода, что даже более важно, чем уменьшение размера определений.

Мур:

> Когда слово является частью чего-то, то оно может быть полезно для придания коду чистоты или для отладки, но никогда оно не будет столь же хорошо, как то, которое всречается много раз. Всякое слово, которое встречается в коде лишь один раз, должно вызывать у Вас желание оценить его значимость. Неоднократно, когда программа становится слишком большой, я возвращаюсь назад и просматриваю ее в поисках фраз, бросающихся в глаза как кандидаты для вычленения. Компьютер не может этого сделать - слишком много вариантов.

При просмотре своей работы Вы часто находите идентичные фразы или короткие словосочетания, повторяющиеся несколько раз. При написании редактора у меня обнаружилась такая повторявшаяся часто фраза:

```
ЭКРАН  КУРСОР @ +
```

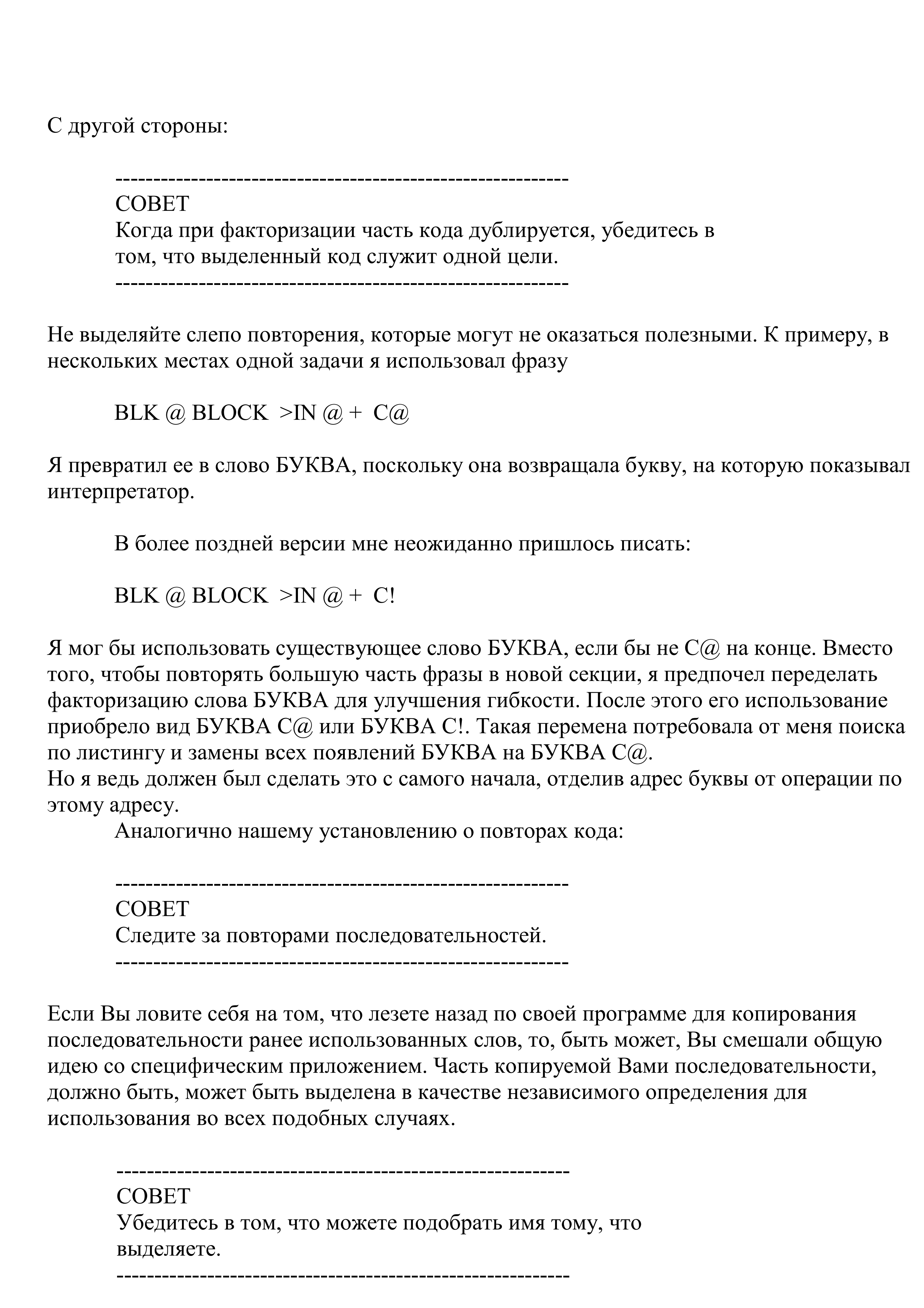
Поскольку она встречалась во многих местах, я выделил ее в отдельное слово НА.

Только от Вас зависит умение выделять изложенные по-разному, но функционально эквивалентные фрагменты, типа:

```
ЭКРАН  КУРСОР @ 1-  +
```

Слово 1- кажется выпадающим из фразы, выделенной в слово НА. На самом же деле это может быть записано как

```
НА 1-
```



С другой стороны:

СОВЕТ
Когда при факторизации часть кода дублируется, убедитесь в том, что выделенный код служит одной цели.

Не выделяйте слепо повторения, которые могут не оказаться полезными. К примеру, в нескольких местах одной задачи я использовал фразу

```
BLK @ BLOCK  >IN @ +  C@
```

Я превратил ее в слово БУКВА, поскольку она возвращала букву, на которую показывал интерпретатор.

В более поздней версии мне неожиданно пришлось писать:

```
BLK @ BLOCK  >IN @ +  C!
```

Я мог бы использовать существующее слово БУКВА, если бы не C@ на конце. Вместо того, чтобы повторять большую часть фразы в новой секции, я предпочел переделать факторизацию слова БУКВА для улучшения гибкости. После этого его использование приобрело вид БУКВА C@ или БУКВА C!. Такая перемена потребовала от меня поиска по листингу и замены всех появлений БУКВА на БУКВА C@.
Но я ведь должен был сделать это с самого начала, отделив адрес буквы от операции по этому адресу.

Аналогично нашему установлению о повторах кода:

СОВЕТ
Следите за повторами последовательностей.

Если Вы ловите себя на том, что лезете назад по своей программе для копирования последовательности ранее использованных слов, то, быть может, Вы смешали общую идею со специфическим приложением. Часть копируемой Вами последовательности, должно быть, может быть выделена в качестве независимого определения для использования во всех подобных случаях.

СОВЕТ
Убедитесь в том, что можете подобрать имя тому, что выделяете.

Мур:

Если у Вас есть что-то, чему Вы не в состоянии присвоить единое имя, не название с внутренними тире, а имя, то значит это - не четко сформированная концепция. Возможность присваивания имени является обязательной частью декомпозиции. Естественно, Вы должны глубоко понимать идею.

Сравните такой взгляд на вещи с критериями для декомпозиции модуля, на защиту которых становится структурное проектирование (глава 1). При таком подходе модуль должен был бы проявлять "функциональную связность", которая может меняться при описании его функции единственным и не составным `предложением`. "Атом" Форта - `имя` - детализирован на порядок лучше.

СОВЕТ
Делайте факторизацию определений так, чтобы скрыть детали, которые мгут измениться.

Важность упрятывания информации мы уже видели в предыдущих главах, особенно в отношении предварительного проектирования. Полезно вспомнить об этом критерии и на этапе разработки.

Вот очень короткое определение, которое только то и делает, что упрятывает информацию:

```
: >BODY  ( acf -- apf )  2+ ;
```

Это определение позволяет превращать адрес поля кода (acf) в адрес поля параметров (apf) вне зависимости от действительного строения словарной статьи. Если бы Вы использовали 2+ вместо слова >BODY, то потеряли бы переносимость в случае, если когда-либо перешли бы на Форт-систему, в которой заголовки отделены от тел определений. (Это - одно из слов набора, предложенного Кимом Харрисом и включенного в список экспериментальных расширений в стандарт Форт-83 [3].)

А вот группа определений, которую можно было бы использовать при написании редактора:

```
: ЭКРАН  ( -- a)  SCR @ BLOCK ;
: КУРСОР  ( -- a)  R# ;
: НА  ( -- a)  ЭКРАН  КУРСОР @ + ;
```

Эти слова могут послужить основой для вычислений всех адресов, потребных для работы с текстом. Их использование полностью устраняет зависимость Ваших алгоритмов редактирования от Форт-блоков.

Что же в этом хорошего? Если бы Вы решили (в процессе развития) создать буфер редактирования для предохранения разрушения блока при ошибках пользователя, то Вам достаточно было бы переопределить пару этих слов, может быть, вот так:

```
CREATE ЭКРАН  1024 ALLOT
VARIABLE КУРСОР
```

Ваш остальной код может оставаться нетронутым.

СОВЕТ

Выделяйте вычислительные алгоритмы из слов, индицирующих результаты.

На самом деле этот вопрос должен выясняться при декомпозиции.

Вот пример. Определенное ниже слово, которое читается как "от-людей-к-связям", вычисляет количество коммуникационных связей, возникающих между данным количеством людей в группе.
(Руководителям программистских коллективов полезно знать - число связей колоссально возрастает с каждым новым членом команды.)

```
: ЛЮДИ>СВЯЗИ  ( #людей -- #связей )  DUP 1-  *  2/ ;
```

Это определение производит только вычисление. Вот "пользовательское определение", которое вызывает ЛЮДИ>СВЯЗИ для расчетов, а зетем печатает результат:

```
: ЛЮДЕЙ  ( #людей)
     ." = "  ЛЮДИ>СВЯЗИ .  ." связей" ;
```

Это дает:

2 ЛЮДЕЙ = <u>1 связей</u>

3 ЛЮДЕЙ = <u>3 связей</u>

5 ЛЮДЕЙ = <u>10 связей</u>

10 ЛЮДЕЙ = <u>45 связей</u>

Даже если Вы уверены, что собираетесь производить калькуляцию лишь один раз, только для распечатки, то, поверьте мне, Вы ошибаетесь. Вам обязательно придется вернуться к этому впоследствии и вычленить вычисляющую часть. Может быть, понадобится отображать всю информацию в выровненной справа колонке, или Вы захотите записать все результаты в базу данных - никогда не знаешь, что случится. Но придется выделить ее обязательно, поэтому лучше сделать это сразу. (Не беда, если даже несколько раз Вы сможете без этого обойтись.)

Лучший пример - слово . (точка). Точка отлично годится в 99% случаев, но иногда оказывается, что она слишком много всего делает. Вот что она в действительности собой представляет (в Форте-83):

```
: .  ( n)  DUP ABS 0  <# #S  ROT SIGN  #>  TYPE  SPACE ;
```

Но, предположим, Вам захотелось преобразовать число на стеке в строку ASCII и поместить ее в буфер для дальнейшей распечатки. Точка делает преобразование, но при этом сразу и печатает. Или, предположим, Вы желаете отформатировать печать игральных карт в виде 10Ч (для "десятки червей"). Нельзя использовать точку для печати 10, поскольку она выводит заключительный пробел.

Вот более удачное разбиение, встречающееся в некоторых Форт-системах:

```
: (.)  ( n -- a #)  DUP ABS 0  <# #S  ROT SIGN  #> ;
: .  ( n)  (.) TYPE SPACE ;
```

Другой пример неотделения выходной функции от вычислений можно найти в нашей собственной работе по вычислению римских чисел из главы 4. При примененном нами решении мы не можем записать полученную римскую цифру в буфер или даже выровнять ее по центру поля. (Лучше было бы использовать слово HOLD вместо EMIT.)

Упрятывание информации может также служить причиной и для `не` разбиения. К примеру, если Вы выделяете фразу

```
SCR @ BLOCK
```

в определение

```
: ЭКРАН   SCR @ BLOCK ;
```

то помните, что Вы это делаете только потому, что Вам может захотеться изменить положение экрана при редактировании. Не заменяйте вслепую все появления фразы на новое слово ЭКРАН, поскольку это определение может быть изменено, а ведь обязательно встретятся несколько мест, где Вам действительно нужно лишь SCR @ BLOCK.

СОВЕТ
Если повторяющийся фрагмент кода для некоторых случаев может измениться, а для других - нет, факторизуйте только те случаи, которые подвержены изменениям. Если фрагмент меняется более, чем одним образом, факторизуйте его в более, чем одно определение.

Понимание того, где нужно упрятывать информацию, требует интуиции и опыта. Пройдя на своем веку множество изменений в проектах, Вы хорошо узнаете, какие вещи наиболее вероятным образом будут изменяться в будущем.

Впрочем, никогда нельзя предвидеть все. Было бы бесполезно даже пытаться, как мы увидим в последующем разделе под названием "Итеративный подход при реализации".

СОВЕТ
Упрощайте командный интерфейс, уменьшая количество команд.

Может показаться парадоксальным, но хорошее расчленение может привести к образованию `меньшего количества` имен. В главе 5 мы видели, как шесть простых имен (ЛЕВЫЙ, ПРАВЫЙ, МОТОР, МАГНИТ, ПУСК, СТОП) выполняли работу восьми плохо отфакторизованных, многочленных имен.

Вот другой пример: на одном предприятии, в котором недавно был внедрен Форт, я обнаружил в ходу два определения. Их назначение было чисто вспомогательным, просто для напоминания программисту, какой словарь - CURRENT, а какой - CONTEXT:

```
: .CONTEXT   CONTEXT @  8 - NFA ID. ;
```

```
: .CURRENT   CURRENT @  8 - NFA ID. ;
```

Если Вы набирали

```
.CONTEXT
```

то система отвечала

.CONTEXT FORTH

(Они работали, - во всяком случае, на использовавшейся здесь системе - возвращаясь назад к полю имени определения словаря и распечатывая его.)

Очевидное повторение кода бросилось мне в глаза в качестве признака плохой факторизации. Можно было бы выделить повторяющийся участок в третье определение:

```
: .СЛОВАРЬ  ( указатель)  @  8 -  NFA ID. ;
```

укорачивая при этом первоначальные определения до

```
: .CONTEXT   CONTEXT .СЛОВАРЬ ;
: .CURRENT   CURRENT .СЛОВАРЬ ;
```

Но при таком подходе единственной разницей между определениями был бы используемый указатель. Поскольку частью хорошей факторизации является уменьшение, а вовсе не увеличение количества определений, казалось логичным иметь всего одно такое определение и позволить ему получать в качестве аргумента либо слово CONTEXT, либо CURRENT.

Применяя принципы выбора подобающих имен, я предложил:

```
: ЕСТЬ  ( адр)  @  8 -  NFA ID. ;
```

для синтаксиса

CONTEXT ЕСТЬ ASSEMBLER

или

CURRENT ЕСТЬ FORTH

Начальная предпосылка этому была в повторении кода, но конечный результат родился в результате попытки упрощения командного интерфейса.

Вот другой пример. В IBM PC имеется четыре чисто текстовых режима отображения:

40 символов монохромно
40 символов в цвете
80 символов монохромно
80 символов в цвете

В моей рабочей Форт-системе имеется слово MODE. Оно берет аргумент в пределах от 0 до 3 и соответственно устанавливает текстовый режим. Разумеется, фразы типа 0 MODE или 1 MODE нисколько не помогали мне запомнить, где какой режим.

Поскольку при работе мне было нужно переключаться между этими режимами, то понадобилось заполучить набор слов для выполнения этой задачи. Слова заодно должны были устанавливать значение переменной, показывающей количество символов - 40 или 80.

Вот самый прямолинейный путь для выполнения этих требований:

```
: 40-Ч/Б           40 #СИМВОЛОВ !  0 MODE ;
: 40-ЦВЕТНОЙ       40 #СИМВОЛОВ !  1 MODE ;
: 80-Ч/Б           80 #СИМВОЛОВ !  2 MODE ;
: 80-ЦВЕТНОЙ       80 #СИМВОЛОВ !  3 MODE ;
```

Производя факторизацию для устранения повторов, мы приходим к такой версии:

```
: СИМВ-РЕЖИМ!  ( #символов режим)  MODE #СИМВОЛОВ ! ;
: 40-Ч/Б           40 0 СИМВ-РЕЖИМ! ;
: 40-ЦВЕТНОЙ       40 1 СИМВ-РЕЖИМ! ;
: 80-Ч/Б           80 2 СИМВ-РЕЖИМ! ;
: 80-ЦВЕТНОЙ       80 3 СИМВ-РЕЖИМ! ;
```

Но, пытаясь уменьшить количество команд, а также следуя принципу нежелательности начинающихся с цифр и сложных, связанных через тире имен, мы обнаруживаем, что можем использовать количество символов как стековый аргумент и `вычислять` режим:

```
: Ч/Б        ( #символов)  DUP #СИМВОЛОВ !  20 / 2-     MODE ;
: ЦВЕТНОЙ    ( #символов)  DUP #СИМВОЛОВ !  20 / 2-  1+ MODE ;
```

Это дает нам такой синтаксис:

```
40 Ч/Б
40 ЦВЕТНОЙ
80 Ч/Б
80 ЦВЕТНОЙ
```

Мы уменьшили количество команд с четырёх до двух.

И снова у нас образовался повторяющийся код. Если его выделить, то получится:

```
: СИМВ-РЕЖИМ!    ( #символов цвет?)
    SWAP  DUP #СИМВОЛОВ !  20 / 2-  + MODE ;
: Ч/Б        ( #символов -- )  0 СИМВ-РЕЖИМ! ;
: ЦВЕТНОЙ    ( #символов -- )  1 СИМВ-РЕЖИМ! ;
```

Мы пришли к более приятному синтаксису, при этом значительно сократив размеры объектного кода. Имея лишь две команды, как в приведенном примере, можно получить ограниченный выигрыш. При большем наборе команд прибыль возрастает геометрически.

Наш последний пример является набором слов для представления цветов в конкретной системе. Имена типа СИНИЙ и КРАСНЫЙ использовать приличнее, чем числа. Одним из решений может быть определение:

```
0 CONSTANT ЧЕРНЫЙ          1 CONSTANT СИНИЙ
2 CONSTANT ЗЕЛЕНЫЙ         3 CONSTANT ЖЕЛТЫЙ
4 CONSTANT КРАСНЫЙ         5 CONSTANT ФИОЛЕТОВЫЙ
```

```
6 CONSTANT КОРИЧНЕВЫЙ            7 CONSTANT СЕРЫЙ
8 CONSTANT ТЕМНО-СЕРЫЙ           9 CONSTANT СВЕТЛО-СИНИЙ
10 CONSTANT СВЕТЛО-ЗЕЛЕНЫЙ       11 CONSTANT СВЕТЛО-ЖЕЛТЫЙ
12 CONSTANT СВЕТЛО-КРАСНЫЙ       13 CONSTANT СВЕТЛО-ФИОЛЕТОВЫЙ
14 CONSTANT СВЕТЛО-КОРИЧНЕВЫЙ        15 CONSTANT БЕЛЫЙ
```

Эти цвета могут использоваться с такими словами, как ФОН, ПЕРО, БОРДЮР:

```
БЕЛЫЙ ФОН  КРАСНЫЙ ПЕРО  СИНИЙ БОРДЮР
```

Однако такое решение требует введения 16-ти имен, многие из которых - сложно-составные. Есть ли путь к упрощению?

Отмечаем, что цвета между 8 и 15 суть "светлые" версии цветов от 0 до 7. (На аппаратном уровне единственным различием между двумя этими наборами является установка бита "интенсивности".) Если мы выделим "светлость", то можем получить такое решение:

```
VARIABLE 'СВЕТ?  ( бит интенсивности?)
: ЦВЕТ  ( цвет)  CREATE ,
   DOES>  ( -- цвет )  @ 'СВЕТ? @  OR  0 'СВЕТ? ! ;
0 ЦВЕТ ЧЕРНЫЙ            1 ЦВЕТ СИНИЙ
2 ЦВЕТ ЗЕЛЕНЫЙ           3 ЦВЕТ ЖЕЛТЫЙ
4 ЦВЕТ КРАСНЫЙ           5 ЦВЕТ ФИОЛЕТОВЫЙ
6 ЦВЕТ КОРИЧНЕВЫЙ            7 ЦВЕТ СЕРЫЙ
: СВЕТЛО   8 'СВЕТ? ! ;
```

При таком синтаксисе слово

```
СИНИЙ
```

само по себе будет возвращать на стеке "1", а фраза

```
СВЕТЛО СИНИЙ
```

будет давать "9". (Приставка-прилагательное СВЕТЛО устанавливает флаг, который используется цветами, а затем очищается.)

При необходимости мы все равно можем определить:

```
8 ЦВЕТ ТЕМНО-СЕРЫЙ
14 ЦВЕТ ГУСТО-ЖЕЛТЫЙ
```

И вновь этот подход привел нас к более приятному синтаксису и более короткому объектному коду.

СОВЕТ

Не расчленяйте ради расчленения. Используйте клише.

Фраза

OVER + SWAP

часто наблюдается в определениях. (Она преобразует адрес и счетчик в конечный и начальный адрес для цикла DO LOOP.)

Другая столь же частая фраза – это

1+ SWAP

(Она преобразует последовательность из начала-счета и конца-счета в последовательность конец-отсчета-плюс-один и начало-отсчета, требуемую для цикла DO LOOP.)

Мало проку в выделении этих фраз в слова, такие, как ДИАПАЗОН (RANGE) (для первой фразы).

Мур:

> Часто встречающаяся фраза (OVER + SWAP) принадлежит к тому виду фраз, которые находятся на грани представимости их в виде полезных слов. Часто, если Вы определили что-то в виде слова, оказывается, что оно встречается лишь однажды.
> Если Вы на свою голову дали имя такой фразе, то Вам придется помнить, что именно делает ДИАПАЗОН. Манипуляции не видны из названия. OVER + SWAP имеет более значимую мнемоническую ценность, чем ДИАПАЗОН.

Я называю такие фразы "клише". Они сами по себе образуют исполненную значения функцию. Не нужно помнить, как работает такая фраза, а только что она делает. И не надо запоминать еще одно имя.

ФАКТОРИЗАЦИЯ ПРИ КОМПИЛЯЦИИ

В последнем разделе мы рассмотрели много способов для организации кода и данных с целью уменьшения избыточности.

Мы можем также применить принцип ограничения избыточности для времени компиляции, позволяя Форту делать за нас черную работу.

> --
> СОВЕТ
> Для получения максимальной управляемости снижайте избыточность даже при компиляции.
> --

Предположим, в задаче требуется нарисовать девять квадратиков, как показано на рис. 6-1.

Рис.6-1. Что нам нужно изобразить.

```
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********


********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********


********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
********    ********    ********
```

Для работы нам нужны константы, представляющие такие значения, как размеры каждого квадратика, зазоры между ними и координаты вершины первого из них.

Разумеется, мы можем определить:

```
8 CONSTANT ШИРИНА
5 CONSTANT ВЫСОТА
4 CONSTANT ПЕРЕУЛОК
2 CONSTANT УЛИЦА
```

(Улицы идут на восток и запад, переулки простираются к северу и югу.)

Теперь мы можем посчитать в уме левый отступ. Мы собираемся отцентрировать все эти квадратики на экране шириной в 80 колонок. Чтобы что-нибудь выровнять, нам надо вычесть его ширину из 80 и поделить пополам, получая при этом левый отступ. Для расчета общей ширины мы складываем:

8 + 4 + 8 + 4 + 8 = 32

(три ширины и два переулка). (80 - 32) / 2 = 24.

Так мы можем определить

```
24 CONSTANT ЛЕВЫЙ-ОТСТУП
```

и проделать то же для верхнего-отступа.

Однако что будет, если позже мы перепроектируем задачу, в результате чего ширина изменится или переулки расширятся? Нам придется вручную пересчитывать отступы.

В среде Форта мы имеем возможность использовать всю силу языка даже при компиляции. Почему бы не позволить Форту самому проделать расчеты?

```
ШИРИНА 3 *  ПЕРЕУЛОК 2 * + 80 SWAP - 2/  CONSTANT ЛЕВЫЙ-ОТСТУП
ШИРИНА 3 *  УЛИЦА 2 * +  24 SWAP - 2/  CONSTANT ПРАВЫЙ-ОТСТУП
```

СОВЕТ
Если значение константы зависит от величины ранее введенной константы, используйте Форт для расчета этого значения.

Все эти вычисления не производятся при работе задачи, так что они не влияют на скорость исполнения.

Вот еще один пример. Рисунок 6-2 содержит код для определения слова, рисующего картинки. Слово РИСОВАТЬ ставит звездочку на каждой координате, указанной в таблице по имени ТОЧКИ. (Здесь слово XY позицирует курсор в позицию (x y), снимаемую со стека.)

Рис.6-2. Еще один пример снижения избыточности при компиляции.

```
: P  ( x y -- )  C, C, ;
CREATE ТОЧКИ
    10 10 P     10 11 P     10 12 P     10 13 P     10 14 P
    11 10 P     12 10 P     13 10 P     14 10 P
    11 12 P     12 12 P     13 12 P     14 12 P
HERE ТОЧКИ -  ( /таблицу)  2/  CONSTANT #ТОЧЕК
: @ТОЧКА  ( i -- x y)  2* ТОЧКИ +  DUP 1+ C@  SWAP C@ ;
: РИСОВАТЬ   #ТОЧЕК 0 DO  I @ТОЧКА  XY  ASCII * EMIT  LOOP ;
```

Обратите внимание на строку, следующую сразу за списком точек:

```
HERE ТОЧКИ -  ( /таблицу)  2/  CONSTANT #ТОЧЕК
```

Фраза "HERE ТОЧКИ -" вычисляет количество байтов, занимаемых таблицей. Деля это значение на два, получаем число координат x-y в списке; это число становится константой #ТОЧЕК, используемой в качестве предела в цикле DO LOOP слова РИСОВАТЬ.

Такая конструкция позволяет Вам добавлять или убирать точки из таблицы, не заботясь о том, сколько их получается. Форт подсчитывает это Вам.

ФАКТОРИЗЦИЯ ПРИ КОМПИЛЯЦИИ С ПОМОЩЬЮ ОПРЕДЕЛЯЮЩИХ СЛОВ.

Давайте исследуем набор подходов к одной и той же проблеме - определению группы связанных адресов. Вот первая попытка:

```
HEX 01A0 CONSTANT БАЗОВЫЙ.АДРЕС.ПОРТА
БАЗОВЫЙ.АДРЕС.ПОРТА CONSTANT ДИНАМИК
БАЗОВЫЙ.АДРЕС.ПОРТА 2+ CONSTANT ПЛАВНИК-А
БАЗОВЫЙ.АДРЕС.ПОРТА 4 + CONSTANT ПЛАВНИК-Б
БАЗОВЫЙ.АДРЕС.ПОРТА 6 + CONSTANT ПОДСВЕТ
DECIMAL
```

Замысел правильный, но реализация его уродлива. Единственные элементы, меняющиеся от порта к порту - это числовые смещения и определяемые имена; все остальное повторяется. Такой повтор подсказывает применение определяющего слова.

Следующий, более читабельный вариант, выделяет весь повторяющийся код в часть "does" определяющего слова:

```
: ПОРТ  ( смещение -- )  CREATE ,
    DOES>  ( -- 'порта )  @  БАЗОВЫЙ.АДРЕС.ПОРТА + ;
0 ПОРТ ДИНАМИК
2 ПОРТ ПЛАВНИК-А
4 ПОРТ ПЛАВНИК-Б
6 ПОРТ ПОДСВЕТ
```

При таком решении мы производим расчет смещения во время `исполнения` всякий раз, когда вызываем эти имена. Было бы более эффективно производить вычисления во время компиляции, например, так:

```
: ПОРТ  ( смещение -- )  БАЗОВЫЙ.АДРЕС.ПОРТА + CONSTANT ;
    \ does>  ( -- 'порта )
0 ПОРТ ДИНАМИК
2 ПОРТ ПЛАВНИК-А
4 ПОРТ ПЛАВНИК-Б
6 ПОРТ ПОДСВЕТ
```

Мы здесь ввели определяющее слово ПОРТ, которое имеет уникальное поведение во время `компиляции` - а именно добавляет смещение к БАЗОВЫЙ.АДРЕС.ПОРТА и определяет константу.

Мы можем даже пройти еще один шаг вперед. Предположим, что адреса всех портов отстоят друг от друга на 2 байта. В этом случае нет такой причины, по которой мы обязаны были бы определять эти смещения. Числовая последовательность

```
0 2 4 6
```

сама по себе избыточна.

В следующей версии мы начинаем с того, что имеем на стеке БАЗОВЫЙ.АДРЕС.ПОРТА. Определяющее слово ПОРТ дублирует адрес, делает из него константу, а затем добавляет 2 к остающемуся на стеке адресу для использования следующим вызовом слова ПОРТ.

```
: ПОРТ  ( 'порта -- 'след-порта )  DUP CONSTANT  2+ ;
    \ does>  ( -- 'порта )
БАЗОВЫЙ.АДРЕС.ПОРТА
    ПОРТ ДИНАМИК
    ПОРТ ПЛАВНИК-А
    ПОРТ ПЛАВНИК-Б
    ПОРТ ПОДСВЕТ
DROP  ( адрес-порта)
```

Отметьте, что мы обязаны дать начальный адрес на стек перед определением первого порта, а затем, после окончания определения всех портов, вызвать DROP для удаления все еще остающегося на стеке адреса.

И последнее замечание. Очень похоже, что базовый адрес порта может меняться, и поэтому должен быть определен в единственном месте. Это `не` означает, что его надо делать константой. Зная, что такой адрес не будет использоваться вне лексикона для имен портов, нисколько не будет хуже дать его здесь просто числом.

```
HEX 01A0  ( базовый адрес портов)  DECIMAL
    ПОРТ ДИНАМИК
    ПОРТ ПЛАВНИК-А
    ПОРТ ПЛАВНИК-Б
    ПОРТ ПОДСВЕТ
DROP
```

ИТЕРАТИВНЫЙ ПОДХОД ПРИ РЕАЛИЗАЦИИ

Ранее в этой книге мы описывали итеративный подход, уделяя особенное внимание его вкладу на стадии проектирования. Поскольку теперь мы рассматриваем реализацию, давайте убедимся, что этот подход в действительности используется и при написании кода.

СОВЕТ
Работайте каждый раз только над одной стороной задачи.

Предположим, в нашу задачу входит рисование или стирание квадрата на заданной координате x-y. (Это та же задача, которую мы предлагали в разделе с названием "Факторизация при компиляции".)

Вначале мы фокусируем свое внимание на рисовании квадратика, не думая о том, как его будем стирать. Вот к чему мы могли бы прийти:

```
: СЛОЙ   ШИРИНА  0 DO  ASCII * EMIT  LOOP ;
: КВАДРАТ  ( верх-левый-x  верх-левый-y -- )
    ВЫСОТА  0 DO  2DUP  I +  XY СЛОЙ  LOOP  2DROP ;
```

Проверив, что это работает правильно, мы переходим теперь к задаче использования этого кода для `стирания` квадратика. Решение просто: вместо того, чтобы вставлять внутрь программы ASCII * мы хотели бы заменять выдаваемый символ со звездочки на пробел. Это требует добавления переменной и некоторых удобочитаемых слов для установки содержимого этой переменной.
Итак:

```
VARIABLE ПЕРО
: РИСОВАТЬ   ASCII * ПЕРО ! ;
: СТИРАТЬ     BL  ПЕРО ! ;
: СЛОЙ   ШИРИНА  0 DO  ПЕРО @ EMIT  LOOP ;
```

Определение КВАДРАТа, так же, как и последующего кода, остается неизменным.

Такой подход дает синтаксис

```
( x y )  РИСОВАТЬ КВАДРАТ
```

или

(x y) СТИРАТЬ КВАДРАТ

Переходя от прямо указанного числа к переменной, содержащей нужной значение, мы добавили уровень подвижности. В нашем случае мы его добавили "задним числом", увеличив уровень сложности слова СЛОЙ без утяжеления определения.

Концентрируясь в данный момент времени на одной проблеме, Вы разрешаете каждое из измерений более эффективно. Если в Ваших размышлениях присутствует ошибка, проблему легче увидеть, если она не загорожена еще не испытанной, неопробованной стороной Вашего кода.

СОВЕТ
Не меняйте слишком многое за один раз.

При редактировании программы - добавлении новой функции или исправлении чего-нибудь - часто хочется взять и подправить одновременно еще несколько других мест. Наш совет: так не делайте.

В каждом своем цикле редактирования-компиляции производите столь мало изменений, сколько только можете. Не забывайте проверять результаты каждой ревизии перед продолжением. Поразительно, насколько часто бывает достаточно проделать всего три невинных исправления только для того, чтобы, перекомпилировав код, убедиться в том, что ничего не работает!

Внесение изменений по одному за раз дает гарантии того, что, когда что-то перестанет работать, Вы будете знать причину.

СОВЕТ
Не пытайтесь слишком рано превзойти все пути для факторизации.

Кое-кто удивляется, почему большинство Форт-систем не имеют определяющего слова МАССИВ (ARRAY). Причина состоит в вышеприведенном правиле.

Мур:

Мне часто попадается тот класс вещей, которые называются массивами. Простейший массив просто добавляет ссылку к адресу и возвращает Вам назад адрес. Массив можно определить фразой

```
CREATE X   100 ALLOT
```

и затем использовать

```
X +
```

Или можно сказать

```
: X   X + ;
```

Одним из наиболее расстраивающих меня вопросов является такой: стоит ли создавать определяющее слово для некоторой структуры данных? Достаточно ли у меня будет оправданий для его существования?
Я редко сразу представляю себе, понадобится ли мне более, чем один массив. Поэтому слово МАССИВ я не определяю. После того, как обнаруживается, что нужно два массива, вопрос становится спорным.
Если нужно три, все ясно. Если только они не разные. И плохо дело, если они разные. Может понадобиться, чтобы массив выдавал свое содержимое. Может хотеться иметь байтовый массив или битовый массив. Может возникнуть желание проверять допустимые границы или помнить его текущую длину для того, чтобы наращивать его на конце.
Стиснув зубы, я спрашиваю себя: "Обязан ли я представлять байтовый массив через двухбайтовый только для того, чтобы подогнать структуру данных к уже имеющемуся у меня слову?"
Чем сложнее проблема, тем менее вероятно, что Вы найдете универсальную структуру данных. Число случаев, когда такая сложная структура может найти широкое применение, очень невелико. Одним из положительных примеров такой сложной организации данных является словарь Форта. Очень крепкая и подвижная структура. Она используется кругом в Форте. Но это - редкость.
Если Вы останавливаетесь на том, чтобы определить слово МАССИВ, то совершаете этим акт декомпозиции. Вы выделяете концепцию массива изо всех слов, в которые впоследствии он войдет. И Вы переходите на следующий уровень абстракции. Построение уровней абстракции - это динамический процесс, Вам его не предугадать.

СОВЕТ
Пусть сегодня это заработает. Оптимизируйте это завтра.

Опять Мур. Во время этого интервью Мур завершал работу над проектированием Форт-компьютера одноплатного уровня с использованием доступных промышленных ИС. В качестве рабочего инструмента для проектирования платы он создал ее симулятор на Форте - для проверки ее логики:

Сегодня утром я понял, что смешиваю описание микросхем с их расположением на плате. Это вполне подходит для моих сегодняшних целей, однако если я зайду на другую плату и захочу использовать для нее те же микросхемы, то для этого вещи сгруппированы плохо.
Мне следовало бы факторизовать их по описаниям в одном месте и по использованию в другом. Тогда я получил бы язык описания микросхем. Хорошо. В то время, когда это делалось, я не был заинтересован в таком уровне оптимизации. Даже если бы прямо тогда мне пришла в голову эта мысль, мне, наверное, нужно было бы сказать: "Все в порядке, я сделаю это попозже", и затем идти вперед так, как я и шел.

Оптимизация тогда не была для меня самым важным. Конечно, я пытаюсь как следует делать факторизацию. Но если хорошего пути не видно, я говорю: "Пусть оно хотя бы работает".
Причина этого не в лени, а в знании того, что объявятся и другие вещи, которые повлияют на решение непредвиденным образом. Попытки оптимизировать все прямо сейчас глупы.
Пока я не увижу перед собой полной картины, я не буду знать, где лежит оптимум.

Наблюдения, приведенные в этом разделе, не должны противопоставляться тому, что было сказано ранее об упрятывании информации и выделении элементов, подверженных изменениям. Хороший программист постоянно пытается балансировать между тягой к встраиванию способностей к изменениям внутрь и склонностью к проведению изменений позже, при необходимости.

Принятие таких решений требует определенного опыта. Но, в качестве общего правила:

СОВЕТ
Противодействуйте вещам-которые-могут-меняться с помощью организации информации, а не наращивания сложности. Добавляйте сложность только при необходимости заставить работать текущую итерацию.

ИТОГИ

В этой главе мы обсуждали различную технику и критерии для факторизации. Мы изучали также итеративный подход в применении к фазе реализации.

ЛИТЕРАТУРА

1. W.P. Stevens, G.J. Myers and L.L. Constantine, `IBM Systems Journal`, vol. 13, no. 2, 1974. Copyright 1974 by International Business Machines Corporation.
2. G.A. Miller, "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information," `Psychol. Rev.`, vol. 63, pp. 81-97, Mar. 1956.
3. Kim R. Harris, "Definition Field Address Conversion Operators," `FORTH-83 Standard`, FORTH Standards Team.

ГЛАВА 7

РАБОТА С ДАННЫМИ :

СТЕКИ И СОСТОЯНИЯ

Форт оперирует с данными одним из двух способов: либо на стеке, либо в структурах данных. Когда какой из подходов применять и как управлять и стеком, и структурами данных - вот тематика этой главы.

ШИКАРНЫЙ СТЕК

Для Форт-слов простейший способ прередачи аргументов друг другу - это через стек. Процесс "прост", поскольку вся работа по заталкиванию и извлечению чисел со стека подразумевается сама собою разумеющейся.

Мур:

> Стек данных использует идею о "скрытой информации". Аргументы, которые должны передаваться между подпрограммами, не представляются в вызывающей последовательности. Один и тот же аргумент способен проходить через целую кучу слов совершенно незаметно, даже ниже уровня осторожности, соблюдаемого программистом, просто потому, что его не надо представлять специально.

Вот один из важных результатов такого подхода: аргументы не именованы. Они находятся на стеке, а не в именованных переменных. Такой эффект - одна из причин элегантности Форта. В то же время это также и одна из причин, по которой плохо написанный на Форте код может быть нечитаемым. Давайте исследуем этот парадокс.

Наличие стека сродни местоимениям в языке. Вот пассаж:

Возьми этот дар, оберни его в папиросную бумагу и положи его в футляр.

Обратите внимание, что слово "дар" использовано лишь однажды.
Дар впоследствии упоминается как "он".

Неконкретность конструкции "его" делает русский (или английский) язык более читабельным (если ссылка однозначна). То же и со стеком, неопределенная передача аргументов делает код более понятным. Мы подчеркиваем `процессы`, а не `передачу аргументов` процессам.

Наша аналогия с местоимениями может подсказать, почему плохо написанный на Форте текст может быть таким нечитабельным. В разговорном языке возникают затруднения, когда на слишком много вещей одновременно ссылаются местоимениями.

Сорви обертку и открой футляр. Вынь дар и выброси его.

В этой фразе конфуз возник оттого, что мы использовали "его" для ссылки одновременно на много вещей. Есть два решения по преодолению этой ошибки. Простейшее - это использование реального имени вместо "него":

Сорви обертку и открой футляр. Вынь дар и выброси `футляр`.

Или мы можем ввести слова "первый" и "последний". Однако лучшим решением было бы перепроектирование фразы:

Сорви обертку и открой подарок. Отбрось футляр.

Так и в Форте мы имеем те же наблюдения:

СОВЕТ

Упрощайте код за счет использования стека. Однако не уходите в стек слишком глубоко внутри отдельно взятого определения. Измените планировку, или, как к последней инстанции, обратитесь к именованной переменной.

Некоторые новички в Форте смотрят на стек так же, как гимнаст глядит на трамплин: как на отличное место для того, чтобы на нем скакать. Однако стек предназначен для передачи данных, а не для акробатики.

Так насколько глубоко это "слишком глубоко"? В общем случае три элемента на стеке - это максимум того, чем Вы можете управлять внутри одного определения. (Для арифметики двойной точности каждый "элемент" занимает две позиции в стеке, но они логично воспринимаются за один элемент такими операторами, как 2DUP, 2OVER и т.д.)

В обычном лексиконе стековых операторов ROT - единственный, который дает доступ к третьему элементу на стеке. Кроме слов PICK и ROLL (которые мы позже прокомментируем), нет легкого способа добраться до того, что лежит ниже.

Для продолжения наших аналогий можно предположить, что три элемента в стеке соответствуют трем русским (английским) местоимениям - "это" ("this"), "то" ("that") и "се" ("th'other").

ПЕРЕПРОЕКТИРОВАНИЕ.

Давайте вообразим ситуацию, когда неверно выбранный подход ведет к проблеме беспорядка на стеке. Предположим, мы пытаемся написать определение слова +THRU (см. главу 5, "Организация листингов", "Блоки загрузки глав"). Мы решили, что тело нашего цикла будет

```
... DO  I LOAD  LOOP ;
```

то есть мы поместим LOAD в цикл и будем давать ему индекс цикла, загружая блоки по их абсолютным номерам.

Изначально на стеке мы имеем:

низ верх

где "низ" и "верх" - это `смещения` от BLK.

Мы должны их представить для DO в следующем виде:

верх+1+blk низ+blk

Самой сложной нашей задачей является прибавление значения BLK к обоим смещениям.

Мы уже ступили на неверную дорожку, но еще об этом не догадываемся. Так что давайте продолжать. Мы пытаемся:

```
низ верх
                BLK @
низ верх blk
                SWAP
низ blk верх
                OVER
низ blk верх blk
                 +
низ blk верх+blk
                 1+
низ blk верх+blk+1
                ROT ROT
верх+blk+1 низ blk
                +
верх+blk+1 низ+blk
```

Мы все проделали, но что же это за путаница!

Если мы любим самобичевание, то должны проделать еще два таких усилия, приходя к

```
BLK @  DUP ROT + 1+  ROT ROT +
```

и к

```
BLK @  ROT OVER +  ROT ROT + 1+  SWAP
```

Все три предложения делают одно и то же, но код, кажется, становится лишь все мрачнее, но не лучше.

Опыт нам подсказывает, что комбинация ROT ROT - это опасный признак: на стеке - столпотворение. Нет необходимости в тщательной проработке вариантов для определения проблемы: как только мы создали две копии "blk", мы сразу получили четыре элемента на стеке.

Первое, о чем обычно вспоминают в этом месте - это о стеке возвратов:

```
BLK @  DUP >R  + 1+  SWAP R> +
```

(Смотрите далее "Шикарный стек возвратов".) Мы здесь продублировали "blk", сохраняя его копию на стеке возвратов и прибавляя другую копию к "верху".

Признаем улучшение. А как читабельность?

Дальше мы думаем: "Может быть, нам нужна именованная переменная." Конечно, у нас уже есть одна: BLK. Поэтому пробуем:

```
BLK @  + 1+  SWAP  BLK @ +
```

Теперь это читается лучше, но все еще слишком длинно, и к тому же избыточно. BLK @ + появляется здесь дважды.

"BLK @ +"? Это звучит как-то знакомо. Наконец наши нейроны дают нужное соединение. Мы смотрим вновь на только что определенное слово +LOAD:

```
: +LOAD  ( смещение -- )  BLK @ +  LOAD ;
```

Это слово, +LOAD, должно было бы и делать такую работу. Все, что нам нужно написать - это:

```
: +THRU  ( низ верх )  1+ SWAP  DO  I +LOAD  LOOP ;
```

Таким образом мы не создали более эффективной версии, поскольку работа BLK @ + будет проделываться в каждом проходе цикла. Однако у нас получился более чистый, более простой концептуально и более читабельный кусок кода. В данном случае неэффективность незаметна, поскольку проявляется один раз при загрузке каждого блока.

Перепроектирование или переосмысливание задачи должно быть тем путем, который нам следует избирать всякий раз, когда дела становятся плохи.

ЛОКАЛЬНЫЕ ПЕРЕМЕННЫЕ.

Большинство задач могут быть перегруппированы так, что лишь несколько аргументов на стеке нужны для них одновременно. Однако бывают ситуации, когда ничего сделать не удается.

Вот пример наихудшего случая. Пусть у нас имеется слово ЛИНИЯ, которое проводит линию между двумя точками, определенными их координатами в следующем порядке:

```
( x1 y1 x2 y2 )
```

где x1,y1 представляют координаты одного конца, а x2,y2 - противоположного конца линии.

Теперь Вам надо написать слово по имени [РАМКА], которое берет четыре аргумента в таком порядке:

```
( x1 y1 x2 y2 )
```

где x1 y1 представляют координаты верхнего левого угла, а x2 y2 - нижнего правого угла рамки.

У Вас не просто четыре элемента на стеке, но каждый из них должен использоваться более одного раза при рисовании линий от точки до точки. Хотя мы и используем стек для получения этих четырех аргументов, алгоритм рисования сам по себе не соответствует природе стека. Если Вы торопитесь, то, быть может, лучшим выходом было бы использовать простое решение:

```
VARIABLE ВЕРХ    ( координата y - вершина рамки)
VARIABLE ЛЕВО    (      "    x - левая сторона)
VARIABLE НИЗ     (      "    y - низ рамки)
VARIABLE ПРАВО (      "    x - правая сторона)
: [РАМКА]   ( x1 y1 x2 y2 )   НИЗ !  ПРАВО !  ВЕРХ !  ЛЕВО !
        ЛЕВО @ ВЕРХ @  ПРАВО @  ВЕРХ @  ЛИНИЯ
        ПРАВО @ ВЕРХ @  ПРАВО @ НИЗ @  ЛИНИЯ
        ПРАВО @ НИЗ @  ЛЕВО @ НИЗ @  ЛИНИЯ
        ЛЕВО @ НИЗ @  ЛЕВО @ ВЕРХ @  ЛИНИЯ ;
```

Мы сделали так: создали четыре именованные переменные, по одной на каждую из координат. Первое, что делает [РАМКА] – это записывает в эти переменные аргументы со стека. Затем с их использованием рисуются линии. Подобные переменные, которые используются лишь внутри определения (или, в некоторых случаях, внутри лексикона) называются "локальными".

Каюсь, что много раз я отчаянно пытался проделать на стеке как можно больше вместо того, чтобы определять локальную переменную. Есть три причины на то, чтобы подобного избегать.

Во-первых, это болезненно сказывается на коде. Во-вторых, результат получается нечитаемым. В-третьих, вся проделанная работа становится бесполезной, когда становится необходимым изменение в проекте, и изменяется порядок следования пары аргументов. Все эти DUPы, ROTы и OVERы в действительности не решили проблему, а только гарцевали вокруг да около.

Имея в виду эту третью причину, я рекомендую следующее:

СОВЕТ

В первую очередь в фазе проектирования держите на стеке только те аргументы, которые используете немедленно. В остальных случаях создавайте локальные переменные. (При необходимости убирайте переменные на стадии оптимизации.)

В-четвертых, если определение чрезвычайно критично ко времени исполнения, подобные заковыристые манипуляции со стеком (типа ROT ROT) могут хорошо поедать тактовые циклы. Прямой доступ к переменным быстрее.

Если это `действительно` критично ко времени, Вам может понадобиться все равно преобразовать его в ассемблер. В этом случае все Ваши проблемы со стеком выставляются за дверь, поскольку любые

Ваши данные будут обрабатываться либо в регистрах, либо косвенно через регистры. К счастью, определения с беспорядочнейшими стековыми аргументами часто принадлежат к тем, что пишутся в коде. Наш примитив [РАМКА] как раз в этом ключе. Другой подобный - это CMOVE>.

Конечно, способ, выбранный нами для [РАМКА], мошенничает, по полчаса выуживая нужное на стек, но он, вне всякого сомнения, является наилучшим решением. Что в нем плохо, так это затраты на создание четырех именованных переменных, заголовков и всего остального для них, исключительно для использования в единственной подпрограмме.

(Если Вы для задачи применяете целевую компиляцию, то при этом заголовков в словаре не требуется, и единственной потерей будут 8 байтов в памяти под переменные. В Форт-системах будущего (*) заголовки могут быть в любом случае вынесены в другие страницы памяти; опять же, потери составят только 8 байтов (**).

Позвольте мне повторить: этот пример демонстрирует наихудший случай, и в большинстве Форт-приложений встречается редко. Если слова хорошо факторизованы, то каждое из них спроектировано для того, чтобы делать очень немногое. Слова, делающие мало, обычно требуют и мало аргументов.

В нашем случае мы имеем дело с двумя точками, каждая из которых представлена двумя координатами.

Нельзя ли изменить проектировку? Первое, ЛИНИЯ, может быть, `слишком` примитивный примитив. Она требует четыре аргумента, поскольку способна рисовать линии между двумя любыми точками, и диагонально, если нужно.

При рисовании нашей рамки нам могут понадобиться только строго вертикальные и горизонтальные линии. В таком лучае мы могли бы написать более мощные, но менее специфичные слова ВЕРТИКАЛЬ и ГОРИЗОНТАЛЬ для их изображения. Каждое из них будет требовать только `трех` аргументов: x и y начальной позиции и длину. Факторизация функции упрощает определение слова [РАМКА].

Или мы могли бы обнаружить, что такой синтаксис для пользователя выглядит более натуральным:

```
10 10 НАЧАЛО!  30 30 РАМКА
```

где НАЧАЛО! устанавливает двухэлементный указатель на то место, откуда будет расти рамка (верхний левый угол). Затем "30 30 РАМКА" рисует рамку в 30 единиц высотой и 30 шириной относительно начала.

(*) - уже настоящего
(**) - на самом деле в обоих случаях - 16 байтов, т.е. 8 слов.

Этот подход сокращает количество стековых аргументов для слова РАМКА как для проектной единицы.

СОВЕТ
При определении состава аргументов, передаваемых через
структуры данных, а не через стек, выбирайте те, которые
более постоянны или которые представляют текущее состояние.

ПО ПОВОДУ СЛОВ PICK И ROLL.

Некоторые чудаки любят слова PICK и ROLL. Они их используют для доступа к элементам с любого уровня стека. Мы их не рекомендуем.

С одной стороны, PICK и ROLL побуждают программиста думать о стеке как о массиве, которым тот не является. Если у Вас настолько много элементов на стеке, что нужны PICK и ROLL, то эти элементы должны были бы быть действительно в массиве.

Второе, программист начинает считать возможным обращаться к аргументам, оставленным на стеке определениями более высокого, вызывающего уровня без того, чтобы эти элементы действительно `передавались` в качестве аргументов, что делает определения зависящими от других определений. Это – признак неструктурированности, и это - опасно.

Наконец, позиция элемента на стеке зависит от того, что находится над ним, а число вещей над ним может постоянно меняться. К примеру, если адрес находится у Вас в четвертом элементе стека, то можно написать

4 PICK @

для загрузки его содержимого. Но Вам придется писать

(n) 5 PICK !

поскольку при наличии на стеке "n" адрес теперь перемещается в пятую позицию.
Подобный код трудно читать и еще труднее переделывать.

ДЕЛАЙТЕ СТЕКОВЫЕ РИСУНКИ.

Когда Вам нужно разрешить некую запутанную стековую ситуацию, лучше ее проработать с помощью карандаша и бумаги. Некоторые люди даже изготавливают формы, типа той, что показана на рис. 7-1. Будучи оформленными подобным образом (вместо закорючек на обороте Вашей телефонной книжки), стековые комментарии становятся приятной внешней документацией.

Рис.7-1. Пример стекового комментария.

Word name: CMOVE> Programmer: LPB Date: 9/23/83

Operations:	Stack effects:	Return stack:
/ / / / / / / / / / / /	s d #	
?DUP IF	s d #	
1- DUP >R	s d #-1	#-1
+	s end-of-d	#-1
SWAP	end-of-d s	#-1
DUP	end-of-d s s	#-1
R>	end-of-d s s #-1	
+	end-of-d s end-of-s	
DO	end-of-d	
I C@	end-of-d last-char	
OVER	end-of-d last-char end-of-d	
C!	end-of-d	
1-	next-to-end-of-d	
-1 +LOOP	"	
ELSE	s d	
DROP	s	
THEN	x	
DROP ;		

СОВЕТЫ ПО СТЕКУ

СОВЕТ
Убедитесь в том, что изменения на стеке происходят при любых возможных потоках передачи управления.

В этом стековом комментарии на слово CMOVE> (рис. 7-1), внутренняя скобка очерчивает содержимое цикла DO LOOP. Глубина стека при выходе из цикла та же, что и при входе в него: один элемент. Внутри внешних скобок результат работы на стеке части IF тот же, что и части ELSE: остается один элемент. (Что представляет собой этот оставшийся элемент, значения не имеет, и поэтому обозначается "x" сразу после THEN.)

СОВЕТ
Когда два действия выполняются над одним и тем же числом, выполняйте сначала ту из функций, которая оставляет свой результат глубже на стеке.

Для примера:

```
: COUNT  ( a -- a+1 # )  DUP C@  SWAP 1+  SWAP ;
```

(где вначале Вы вычисляете счетчик) более эффективно может быть записано в виде:

```
: COUNT  ( a -- a+1 # )  DUP 1+  SWAP C@  ;
```

СОВЕТ
Где можно, сохраняйте количество возвращаемых аргументов постоянным во всех возможных случаях.

Часто обнаруживаются определения, которые выполняют некоторую работу и, если что-то было не так, возвращают код ошибки. Вот один из путей, по которому можно спроектировать стековый интерфейс:

(-- код-ошибки f | -- t)

Если значение флага - истина, то операция удачна. Если же оно - ложь, то операция окончилась неудачей и на стеке присутствует еще одно число, показывающее природу ошибки.

Вы можете обнаружить, что манипуляции со стеком проще, если переделать интерфейс так:

(-- код-ошибки | 0=нет-ошибки)

Одно число служит и флагом, и (при неудаче) номером ошибки.

Обратите внимание на то, что при этом использована обратная логика: не равенство нулю говорит об ошибке. Под значение кода может быть использовано любое число, кроме нуля.

ШИКАРНЫЙ СТЕК ВОЗВРАТОВ

А как насчет использования стека возвратов для хранения временных аргументов? Хороший ли это стиль?

Некоторые люди с большой осторожностью используют этот стек. Однако стек возвратов предлагает самое простое разрешение некоторым неприятным стековым ситуациям. Посмотрите на определение CMOVE> из предыдущего раздела.

Если вы решаете использовать стек возвратов для этого, то помните, что Вы при этом используете компонент Форта, предназначенный для иных целей. (Смотрите раздел под названием "Совместное использование компонентов" далее в этой главе.)

Вот некоторые рекомендации о том, как не попасться на собственный крючок:

СОВЕТ

1. Операции со стеком возвратов должны располагаться симметрично.
2. Они должны располагаться симметрично при любых направлениях передачи управления.
3. При факторизации определений следите, чтобы не получилось так, что одна часть разбиения содержит один оператор, работающий со стеком возвратов, а другая часть - его ответный оператор.
4. При использовании внутри цикла DO LOOP такие операторы должны использоваться симметрично внутри цикла, а слово I работает неверно в окружении >R и R>.

Для каждого из слов >R должно присутствовать R> внутри того же определения. Иногда операторы кажутся расположенными симметрично, но из-за извилистого пути передачи управления таковыми не являются. К примеру:

```
... BEGIN  ...  >R  ...  WHILE  ... R>  ...  REPEAT
```

Если такая конструкция используется во внешнем цикле Вашей задачи, все будет в порядке до тех пор, пока Вы из него не выйдете (быть может, через много часов), и тогда все неожиданно зависнет. Причина? При последнем прохождении цикла разрешающий оператор R> не был исполнен.

ПРОБЛЕМА ПЕРЕМЕННЫХ

Хотя немедленно интересующие нас данные мы держим на стеке, но в то же время мы зависим и от большого количества информации, заключенной в переменных, и готовой к частому доступу. Участок кода может изменить содержимое переменной без того, чтобы обязательно знать способ использования этих данных, кто их использует или же когда и будут ли они вообще использоваться. Другой кусок кода может взять содержимое переменной и использовать его без знания того, откуда там это значение.

Для каждого слова, которое кладет значение на стек, другое слово должно снимать это значение. Стек дает нам соединение от точки к точке, наподобие почты.

Переменные, с другой стороны, могут быть установлены любой командой и считаны любое число раз - или совсем не считаны - любой командой. Переменные доступны каждому, кто удосужится на них взглянуть - как картины.

Так, переменные могут использоваться для отображения текущего положения дел. Использование такого отображения может упростить задачу. В примере с римскими цифрами из главы 4 мы использовали переменную #КОЛОНКИ для представления текущего положения смещения; слова ЕДИНИЧКА, ПЯТЕРКА и ДЕСЯТКА зависели от этой информации для определения того, какой тип символа печатать. Нам не пришлось каждый раз применять описания типа ДЕСЯТКИ ЕДИНИЧКА, ДЕСЯТКИ ПЯТЕРКА и т.д.

С другой стороны, такое использование добавляет новый уровень сложности. Для того, чтобы сделать что-то текущим, мы должны определять переменную или некоторый тип структуры данных. Мы также должны помнить о том, чтобы ее инициализировать, если есть вероятность того, что какой-либо кусок кода начнет на нее ссылаться до того, как мы успеем ее установить.

Более серьезной проблемой в переменных является то, что они не "реентерабельны". В многозадачной Форт-системе каждая из задач, которая требует локальных переменных, должна иметь свои собственные копии их. Для этого служат переменные типа USER.
(См. "Начальный курс ...", гл. 9, "География Форта").

Даже в пределах одной задачи определение, ссылающееся на переменную, труднее проверять, изменять и использовать повторно в другой обстановке, отличной от такой, в которой аргументы передаются через стек.

Представим себе, что мы разрабатываем редактор для текстового процессора. Нам нужна программа, которая вычисляет количество символов между текущим положением курсора и предыдущей последовательностью возврат-каретки/перевод-строки. Так мы пишем слово, в котором при помощи цикла DO LOOP, начиная от текущей позиции (КУРСОР @) и заканчивая нулевой позицией, производится поиск символа перевода строки.

Когда в цикле обнаруживается искомая символьная последовательность, мы вычитаем ее относительный адрес из нашего текущего положения курсора

```
ее-позиция КУРСОР @  SWAP -
```

для получения расстояния между ними.

Стековая картинка слова будет:

```
( -- расстояние-до-предыдущего-вк/пс )
```

Но при последующем кодировании мы обнаруживаем, что аналогичное слово нужно для вычисления расстояния от произвольного символа - \`не\` от текущей позиции курсора. Мы останавливаемся на том, что вычленяем "КУРСОР @" и передаем начальный адрес через стек в качестве аргумента, получая:

```
( начальное-положение -- расстояние-до-предыдущего-вк/пс )
```

Выделив ссылку на переменную мы сделали определение более полезным.

СОВЕТ
За исключением случаев, когда манипуляции со стеком
достигают уровня нечитабельности, пытайтесь передавать

аргументы через стек вместо того, чтобы брать их из переменных.

Кожж:

Большая часть модульности Форта происходит от проектирования и понимания слов Форта как "функций" в математическом смысле. Мой опыт показывает, что Форт-программист обычно старается избегать определения любых, кроме наиболее существенных глобальных переменных (у меня есть друг, у которого над столом висит надпись "Помоги убрать переменные"), и пытается писать слова со свойством так называемой "ссылочной переносимости", т.е. при одних и тех же данных на стеке слово всегда дает одинаковый ответ независимо от более общего контекста, в котором оно исполняется.

Это на деле - как раз то свойство, которое мы используем для индивидуальной проверки слов. Слова, не имеющие его, гораздо труднее тестировать. Чувствуется, что "именованные переменные", значения которых часто меняются - вещь, немногим лучшая "запрещенного" ныне GOTO.



Ранее мы предлагали использовать локальные переменные в первую очередь во время фазы проектирования для уменьшения движения на стеке. Важно отметить, что при этом

переменные использовались только внутри одного определения. В нашем примере [РАМКА] получает четыре аргумента со стека и немедленно загружает их в локальные переменные для собственного употребления. Четыре переменные вне определения не используются, и слово работает так же надежно, как функция.

Программисты, не привыкшие к языку, в котором данные могут передаваться автоматически, не всегда используют стек так хорошо, как можно бы. Майкл Хэм выдвигает предположение, что начинаюшие пользователи Форта не доверяют стеку [1]. Он говорит о том, что вначале действительно чувствуешь себя безопасней при хранении чисел в переменных вместо содержания их на стеке. Ощущаешь, что "лучше и не говорить о том, что `могло` бы произойти со стеком во время всей этой кутерьмы".

Хэму потребовалось некоторое время на понимание того, что "если слова правильно заботятся о себе, используя стек только для ожидаемого ввода и вывода и чистя его за собой, то они могут рассматриваться как замкнутые системы ... Я могу положить счетчик на стек в начале цикла, пройти через всю программу в каждой ее части, и в конце нее счетчик будет существовать опять на вершине стека, ни на волос не сместившись".

ЛОКАЛЬНЫЕ И ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ / ИНИЦИАЛИЗАЦИЯ

Как мы уже говорили раньше, переменная, которая используется исключительно внутри одного определения (или одного лексикона), и спрятана от остального кода, называется локальной. Переменная же, используемая более, чем одним лексиконом, называется глобальной. Как мы видели в предыдущей главе, набор глобальных переменных, которые совместно описывают общий интерфейс между несколькими лексиконами, называется "интерфейсным лексиконом".

Форт не делает различий между локальными и глобальными переменными. Их проводят Форт-программисты.

Мур:

> Нам следовало бы писать для читателя. Если на что-то, типа временной переменной для накапливания суммы, ссылаются лишь локально, то мы и должны определить это что-то локально.
> Подручнее определить это в том же блоке, где оно используется, где приведены нужные примечания.
> Если же что-то используется глобально, то мы должны составлять логически взаимосвязанные вещи и определять их все вместе в отдельном блоке. По одной на строку и с комментарием.
> Вопрос: где Вы все это будете инициализировать? Кое-кто считает, что надо делать это в той же строке, сразу же после определения. Но при этом вытесняются комментарии, и не остается места под добавочные примечания. И инициализация разбрасывается по всему тексту задачи.
> Я стараюсь проделывать всю инициализацию в блоке загрузки. После того, как я загрузил все свои блоки, инициализирую все, что должно быть инициализировано. Инициализация может даже включать в себя установку таблиц цветов или вызов какой-нибудь программы сброса.
> Если Ваша программа обречена на целевую компиляцию, то легко вписать сюда же слово, которое производит все установки. При этом можно делать и гораздо

больше работы. Мне случалось давать определения переменных в ПЗУ, причем их тела размещались в массиве в верхней памяти, а начальные значения - в ПЗУ, и я копировал наверх эти значения во время инициализации. Впрочем, чаще надо просто занести в несколько переменных что-нибудь, отличное от нуля.

СОХРАНЕНИЕ И ВОССТАНОВЛЕНИЕ СОСТОЯНИЯ

Переменные характерны тем, что когда меняешь их содержимое, то затираешь значение, которое там было раньше. Давайте рассмотрим некоторые проблемы, которые это может породить и кое-что из того, что можно с ними сделать.

BASE - это переменная, которая показывает текущую системы счисления для всего числового ввода и вывода. Следующие слова обычно присутствуют в Форт-системах:

```
: DECIMAL   10 BASE ! ;
: HEX   16 BASE ! ;
```

Представьте себе, что мы пишем слово, печатающее "дамп" памяти. Обычно мы работаем в десятичном режиме, но хотим показать дамп в шестнадцатеричном. Мы пишем так:

```
: DUMP  ( a # )
    HEX   ...  ( код для дампа )  ...  DECIMAL ;
```

Это работает - большую часть времени. Однако здесь заложена презумпция того, что мы хотим вернуться назад к десятичной системе. А что как если мы работали в шестнадцатеричной и хотим вернуться именно к ней? До изменения значения переменной BASE на HEX нам следует сохранить ее текущее значение. Когда дамп закончится, мы его восстановим.

Это значит, что мы должны на время отложить старое значение, пока формируем дамп. Одно из возможных мест для этого
- стек возвратов:

```
: DUMP  ( a # )
    BASE @ >R  HEX   ( код для дампа )  R> BASE ;
```

Если это покажется сложным, мы можем определить временную переменную:

```
VARIABLE СТАРЫЙ-BASE
: DUMP  ( a # )
    BASE @  СТАРЫЙ-BASE !  HEX  ( код для дампа )
    СТАРЫЙ-BASE @  BASE ! ;
```

Как же быстро все усложняется.

В этой ситуации если и текущее, и старое содержимое переменной принадлежит только Вашей задаче (и не является частью Вашей системы), и если такая ситуация возникает более, чем один раз, примените технику факторизации:

```
: СХОРОНИТЬ  ( a)  DUP 2+  2 CMOVE> ;
```

```
: ЭКСГУМИРОВАТЬ ( а) DUP 2+ SWAP 2 CMOVE ;
```

Затем вместо определения двух переменных типа УСЛОВИЕ и СТАРОЕ-УСЛОВИЕ, определите одну переменную двойной длины:

```
2VARIABLE УСЛОВИЕ
```

Используйте СХОРОНИТЬ и ЭКСГУМИРОВАТЬ для сохранения и восстановления начального значения:

```
: ДЕЛИШКИ        УСЛОВИЕ СХОРОНИТЬ   17 УСЛОВИЕ !       ( делишки)
    УСЛОВИЕ ЭКСГУМИРОВАТЬ ;
```

СХОРОНИТЬ держит "старую" версию условия по адресу УСЛОВИЕ 2+.

Вам все равно следует быть осторожными. Возвращаясь к нашему примеру DUMPа, предположим, что мы решили добавить черту дружелюбия, позволяя пользователю прекращать дамп в любой момент нажатием на клавишу "выход". Итак, внутри цикла мы строим проверку на нажатие клавиши и в при необходимости исполняем QUIT. Но что же при этом случается?

Пользователь начинает работать в десятичной системе, затем он вводит DUMP. Он покидает дамп на полдороге и оказывается, на свое удивление, в шестнадцатеричной.

В простом, подручном случае лучшим решением является использование не QUIT, а контролируемого выхода (через LEAVE и т.д.) в конец определения, где BASE восстанавливается.

В очень сложных задачах контролируемый выход часто непрактичен, хотя множество переменных должно быть как-то восстановлено в своих естественных значениях.

Мур ссылается на такой пример:

> Вы и вправду оказались завязанными в узел. Сами себе создаете трудности. Если мне нужен шестнадцатеричный дамп, я говорю HEX DUMP. Если нужен десятичный, то говорю DECIMAL DUMP. Я не доверяю слову DUMP привилегию произвольно менять мое окружение.
> Имеется философский выбор между восстановлением ситуации по окончании и установкой ситуации в начале. Долгое время мне казалось, что я должен восстанавливать все в конце. И я должен был бы делать это постоянно и везде. Но ведь трудно определить "везде". Так что теперь я стараюсь устанавливать ситацию перед началом.
> Если у меня есть слово, которое отвечает за положение вещей, то лучше самому определить это положение. Если же кто-то другой его изменяет, то и ему не надо беспокоиться о том, чтобы его восстанавливать.
> Всегда имеется больше выходов, чем входов.

В случаях, когда нужно что-то сбросить до окончания работы, мне кажется полезным использование единственного слова (я называю его ОЧИСТКА). Я вызываю слово ОЧИСТКА:

* в точке нормального завершения программы
* там, где пользователь может свободно выйти (перед QUIT)
* перед любой точкой, в которой может возникнуть фатальная ошибка, которая вызывает аварийное завершение (ABORT).

Наконец, если перед Вами возникает ситуация, когда надо сохранять/восстанавливать значение, убедитесь в том, что это не есть следствие плохой факторизации. Предположим, к примеру, что мы написали:

```
: ДЛИННАЯ    18 #ДЫР ! ;
: КОРОТКАЯ    9 #ДЫР ! ;
: ИГРА    #ДЫР @  0 DO  I ДЫРА ИГРАТЬ  LOOP ;
```

Текущая ИГРА может быть либо короткой, либо длинной.

Позже мы решили, что нам нужно слово для игры с `любым` количеством дыр. Поэтому мы вызываем слово ИГРА так, чтобы не испортить текущий тип игры (число дыр):

```
: ДЫРЫ  ( n)  #ДЫР @  SWAP #ДЫР !  ИГРА  #ДЫР ! ;
```

Поскольку нам понадобилось слово ДЫРЫ уже после того, как была определена ИГРА, то, кажется, оно и должно иметь большую сложность: мы строим ДЫРЫ вокруг игры. Но на самом деле – может быть, Вы это уже видите - правильнее будет переделать:

```
: ДЫРЫ  ( n)  0 DO  I ДЫРА ИГРАТЬ  LOOP ;
: ИГРА    #ДЫР @ ДЫРЫ ;
```

Мы можем построить игру вокруг дыр и избежать всей этой белиберды с запоминанием/восстановлением.

ВНУТРЕННИЕ СТЕКИ ПРОГРАММ

В последнем разделе мы исследовали некоторые пути для сохранения и восстановления `единственного` предыдущего значения. Некоторые задачи требуют того же для `нескольких` значений. Часто для Вас может оказаться самым удобным определение своего собственного стека.

Вот исходный текст для пользовательского стека, имеющий очень простой контроль ошибок (при ошибках стек очищается):

```
CREATE СТЕК  12 ALLOT  \ { 2указ-стека | 10стек [5 элем.] }
HERE CONSTANT СТЕК>
: ИНИЦ-СТЕК   СТЕК СТЕК ! ;   ИНИЦ-СТЕК
: ?СБОЙ  ( ?)  IF  ." Ошибка стека"  ИНИЦ-СТЕК ABORT  THEN ;
: PUSH  ( n)  2 СТЕК +!  СТЕК @  DUP  СТЕК> =  ?СБОЙ ;
: POP  ( -- n)  СТЕК @ @  -2 СТЕК +!  СТЕК @  СТЕК < ?СБОЙ ;
```

Слово PUSH берет число со стека данных и "заталкивает" его на этот новый стек. POP работает обратным образом, число "всплывает" из нового стека и ложится на Фортов стек данных.

В реальной задаче Вам может захотеться применить другие имена для PUSH и POP с тем, чтобы они лучше соответствовали своему концептуальному назначению.

СОВЕТ
Можно использовать компонент для дополнительных нужд, кроме тех, для которых он введен, если выполняются такие условия:
1. Все использования компонента взаимно исключают друг друга
2. При каждом использовании компонента с прерыванием предыдущего обращения состояние компонента восстанавливается по окончании использования.

Мы видели простой пример использования этого принципа со стеком возвратов. Этот стек - компонент Форт-системы, служащий для того, чтобы держать адреса возвратов, и вследствие этого указывающий, откуда Вы пришли и куда идете. Можно использовать этот стек и для хранения временных значений, и во многих случаях это хорошо. Проблемы же возникают тогда, когда игнорируется один из вышеуказанных принципов.

В моем форматтере текста вывод может производиться невидимо. Это делается с двумя целями: (1) для заглядывания вперед с целью поиска какого-либо совпадения, и (2) для форматирования списка содержания (весь документ форматируется и вычисляются номера страниц без реального отображения чего бы то ни было).

Было соблазнительно думать, что раз уж я добавил однажды возможность делать невидимым вывод, то мог бы использовать эту возможность для обслуживания обеих названных целей. К несчастью, цели эти взаимно друг друга не исключают.

Давайте посмотрим, что могло бы произойти, если бы я пренебрег этим правилом. Пусть слово ВЫВОД производит вывод, и оно достаточно умно, чтобы знать, видимый он или нет. Слова ВИДИМЫЙ и НЕВИДИМЫЙ ставят нужный режим.

Мой код для заглядывания вперед вначале исполняет слово НЕВИДИМЫЙ, затем проверяет-форматирует дальнейший текст для определения его длины, и в конце исполняет ВИДИМЫЙ для восстановления положения вещей.

Это отлично работает.

Позже я добавляю режим сбора содержания. Вначале код выполняет НЕВИДИМЫЙ, затем проходит по документу, собирая номера страниц и т.д.; и наконец выполняет ВИДИМЫЙ для восстановления нормального вывода.

Понятно? Предположим, что я формирую содержание и дошел до того места, где делается заглядывание вперед. Когда заглядывание кончается, я исполняю ВИДИМЫЙ. И неожиданно я принимаюсь печатать документ в то самое время, когда предполагал составлять содержание.

Каково же решение? Их может быть несколько.

Одно из решений рассматривает проблему в разрезе того, что заглядывающий код анализирует флаг видимости/невидимости, который мог быть предварительно установлен составителем содержания. Таким образом, код для заглядывания должен был бы уметь сохранять и впоследствии восстанавливать этот флаг.

Другое решение предусматривает наличие двух раздельных переменных - одной для индикации того, что мы заглядываем вперед, а другой - для индикации печати содержания. Слово ВЫВОД требует, чтобы оба флага содержали ЛОЖЬ для того, чтобы и вправду что-нибудь напечатать.

Имеются два пути для реализации последнего подхода, в зависимости от того, как Вы хотите произвести декомпозицию задачи. Первое, мы могли бы вложить проверку одного условия в проверку второго:

```
: [ВЫВОД]  ...
      ( исходное определение, всегда производит вывод) ... ;
     VARIABLE 'ЗАГЛЯД?  ( t=заглядывание)
     : <ВЫВОД>  'ЗАГЛЯД? @ NOT IF  [ВЫВОД]  THEN ;
     VARIABLE 'ОГЛ?  ( t=составление-содержания)
     : ВЫВОД   'ОГЛ? @ NOT IF  <ВЫВОД>  THEN ;
```

ВЫВОД проверяет, что мы не делаем составление содержания, и вызывает <ВЫВОД>, который, в свою очередь, проверяет, что мы не заглядываем вперед, и вызывает [ВЫВОД].

В цикле разработки слово [ВЫВОД], которое всегда делает вывод, изначально называлось ВЫВОД. Затем для включения проверки на заглядывание было определено новое слово ВЫВОД, а старое - переименовано в [ВЫВОД], добавляя таким образом уровень сложности задним числом без нарушения какого-либо кода, который использует слово ВЫВОД.

Наконец, по добавлении составления содержания, для этой проверки определился новый ВЫВОД, а предыдущий был переименован в <ВЫВОД>.

Это - один подход к использованию двух переменных. Другой состоит в том, что оба теста делаются в одном слове:

```
: ВЫВОД    'ЗАГЛЯД? @  'ОГЛ? @ OR  NOT IF  [ВЫВОД]  THEN ;
```

Однако в данном конкретном случае еще один подход может упразднить всю эту суету. Мы можем использовать единственную переменную не в качестве флага, а как счетчик.

Определяем:

```
VARIABLE 'НЕВИДИМЫЙ?  ( t=невидимый)
: ВЫВОД   'НЕВИДИМЫЙ? @ 0= IF  [ВЫВОД]  THEN ;
: НЕВИДИМЫЙ   1 'НЕВИДИМЫЙ? +! ;
: ВИДИМЫЙ   -1 'НЕВИДИМЫЙ? +! ;
```

Заглядывающий код начинает с вызова НЕВИДИМЫЙ, который наращивает счетчик на единичку. Значение, отличное от нуля - это "истина" (t), поэтому ВЫВОД не будет работать. По завершению заглядывания код вызывает ВИДИМЫЙ, который уменьшает счетчик назад к нулю ("ложь").

Код составления содержания также начинает с НЕВИДИМЫЙ и заканчивает ВИДИМЫЙ. Когда при сборе содержания мы достигаем заглядывания вперед, второе применение НЕВИДИМЫЙ увеличивает значение счетчика до 2. Последующий вызов ВИДИМЫЙ уменьшает счетчик до единицы, так что мы все равно остаемся невидимыми до тех пор, пока содержание не будет составлено.

(Отметьте, что мы должны применять 0= вместо NOT. В стандарте-83 изменена функция NOT так, что она означает инверсию чего-либо, так что 1 NOT означает "истина". Между прочим, я думаю, что это было ошибкой.)

Такое использование счетчика может быть, однако, опасным. Оно требует парности в использовании команд: два вызова ВИДИМЫЙ приводят к ситуации невидимости. Чтобы этого не было, можно проверять в ВИДИМЫЙ обнуление счетчика:

```
: ВИДИМЫЙ   'НЕВИДИМЫЙ? @ 1-  0 MAX  'НЕВИДИМЫЙ? ! ;
```

ТАБЛИЦА СОСТОЯНИЯ

Одна переменная способна отображать единственное условие либо флаг, значение или адрес функции.

Собранные вместе условия представляют `состояние` задачи или определенного компонента [2]. Некоторые приложения требуют возможности сохранения текущего состояния для его последующего восстановления или, быть может, чтобы иметь альтернативные состояния.

СОВЕТ
Когда для задачи требуется содержать целую группу условий одновременно, используйте таблицу состояния, а не отдельные переменные.

В простейшем случае требуется сохранение и восстановление состояния. Предположим, мы изначально имеем шесть переменных, представляющих определенный компонент, как показано на рис. 7-2.

Рис..7-2. Собрание родственных переменных.

```
VARIABLE СВЕРХУ
VARIABLE СНИЗУ
VARIABLE СЛЕВА
VARIABLE СПРАВА
VARIABLE ВНУТРИ
VARIABLE СНАРУЖИ
```

Теперь предположим, что нам всех их надо сохранить таким образом, чтобы можно было произвести дальнейшие действия, а затем обратно восстановить. Мы могли бы определить:

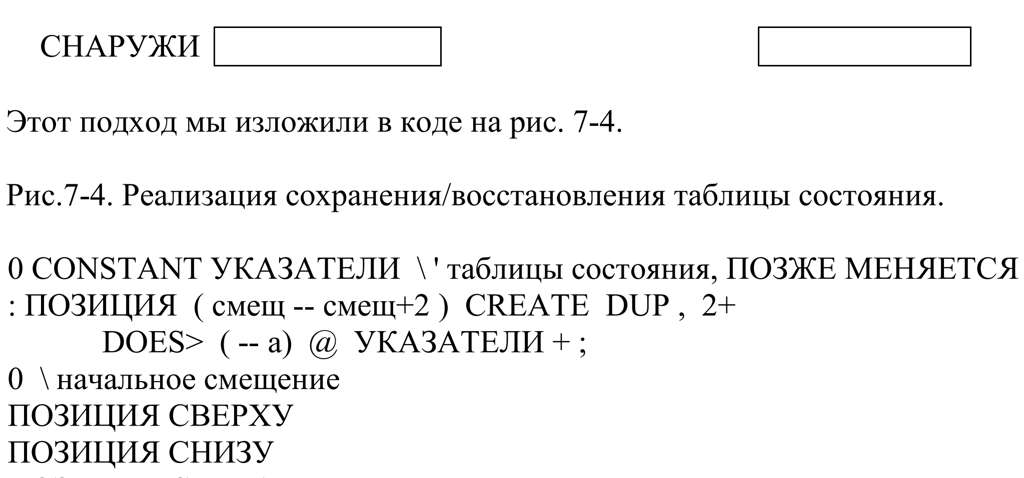
```
: @СОСТОЯНИЕ  ( -- сверху снизу слева справа внутри снаружи)
        СВЕРХУ @  СНИЗУ @  СЛЕВА @  СПРАВА @  ВНУТРИ @  СНАРУЖИ @ ;
: !СОСТОЯНИЕ  ( сверху снизу слева справа внутри снаружи -- )
        СНАРУЖИ !  ВНУТРИ !  СПРАВА !  СЛЕВА !  СНИЗУ !  СВЕРХУ ! ;
```

таким образом сохраняя значения на стеке до тех пор, пока не придет время их восстановить. Либо мы могли бы определить второй набор переменных для всех вышеперечисленных, и в них по отдельности сохранять состояние.

Однако предпочтительней будет такая технология, при которой создается таблица, а каждый элемент таблицы имеет свое имя. Затем определяется вторая таблица такой же длины. Как видно из рисунка 7-3, можно сохранять состояние копированием таблицы, называемой УКАЗАТЕЛИ, во вторую таблицу по имени АРХИВ.

Рис.7-3. Концептуальная модель для сохранения таблицы состояния.

	УКАЗАТЕЛИ		АРХИВ
СВЕРХУ			
СНИЗУ			
СЛЕВА		⇨	
СПРАВА			
ВНУТРИ			



Этот подход мы изложили в коде на рис. 7-4.

Рис.7-4. Реализация сохранения/восстановления таблицы состояния.

```
0 CONSTANT УКАЗАТЕЛИ  \ ' таблицы состояния, ПОЗЖЕ МЕНЯЕТСЯ
: ПОЗИЦИЯ  ( смещ -- смещ+2 )  CREATE  DUP ,  2+
        DOES>  ( -- a)  @  УКАЗАТЕЛИ + ;
0  \ начальное смещение
ПОЗИЦИЯ СВЕРХУ
ПОЗИЦИЯ СНИЗУ
ПОЗИЦИЯ СЛЕВА
ПОЗИЦИЯ СПРАВА
ПОЗИЦИЯ ВНУТРИ
ПОЗИЦИЯ СНАРУЖИ
CONSTANT /УКАЗАТЕЛИ  \ ' конечного вычисленного смещения

HERE ' УКАЗАТЕЛИ >BODY !  /УКАЗАТЕЛИ ALLOT  \ реальная таблица
CREATE АРХИВ  /УКАЗАТЕЛИ ALLOT  \ место для сохранения
: АРХИВИРОВАТЬ     УКАЗАТЕЛИ АРХИВ  /УКАЗАТЕЛИ CMOVE ;
: РЕСТАВРИРОВАТЬ   АРХИВ УКАЗАТЕЛИ  /УКАЗАТЕЛИ CMOVE ;
```

Обратите внимание, что в этой реализации имена указателей - СВЕРХУ, СНИЗУ и т.д. всегда возвращают один и тот же адрес. Для представления текущих значений любых состояний всегда используется лишь одно место в памяти.

Также отметьте, что мы определяем УКАЗАТЕЛИ (имя таблицы) как константу, а не через CREATE, используя для этого подставное нулевое значение. Это делается для того, чтобы ссылаться на УКАЗАТЕЛИ в определяющем слове ПОЗИЦИЯ, иначе мы не можем этого делать, пока не закончим определять имена полей, не выясним реальный размер таблицы и не будем в состоянии выполнить для нее ALLOT.

Когда имена полей созданы, мы определяем размер таблицы как константу /УКАЗАТЕЛИ. Наконец мы резервируем место для самой таблицы, модифицируя ее начальный адрес (HERE) внутри константы УКАЗАТЕЛИ. (Слово BODY> преобразует адрес, возвращаемый словом ', в адрес содержимого константы.) И вот УКАЗАТЕЛИ возвращают адрес определенной позже таблицы так же, как определенное через CREATE слово возвращает адрес таблицы, расположенной немедленно после заголовка этого слова.

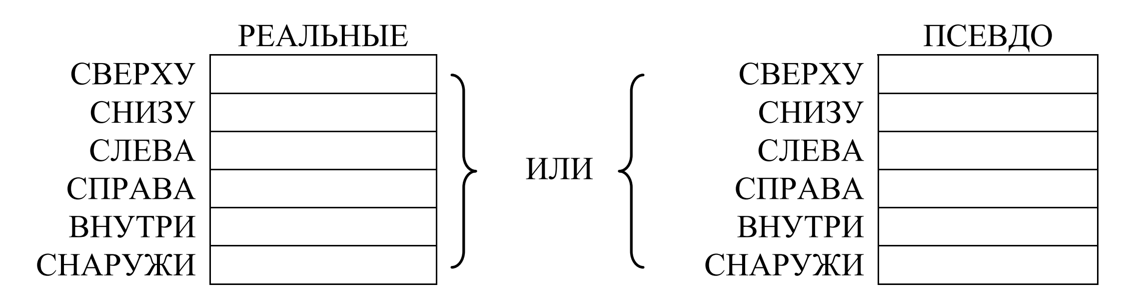
Хотя мы имеем право менять значение константы во время компиляции, как в нашем примере, здесь есть стилистическое ограничение:

СОВЕТ

Значение константы никогда не следует изменять после окончания компиляции задачи.

Случай альтернативных состояний несколько более сложен. В этой ситуации нам приходится переключаться вперед и назад между двумя (или более) состояниями, при этом не перепутывая условия в каждом из состояний. Рис. 7-5 демонстрирует концептуальную модель такого рода таблицы состояния.

Рис.7-5. Концептуальная модель для таблиц альтернативных состояний.



В этой модели имена СВЕРХУ, СНИЗУ и т.д. могут быть выполнены так, что будут указывать на две таблицы, РЕАЛЬНЫЕ и ПСЕВДО. Делая текущей таблицу РЕАЛЬНЫЕ, мы устанавливаем все имена-указатели внутрь этой таблицы; делая текущей таблицу ПСЕВДО, получаем адреса внутри другой таблицы.

Программа на рис. 7-6 реализует механизм альтернативных состояний.

Рис.7-6. Реализация механизма альтернативных состояний.

```
VARIABLE 'УКАЗАТЕЛИ  \ указатель на таблицу состояния
: УКАЗАТЕЛИ  ( -- адр текущей таблицы)  'УКАЗАТЕЛИ @ ;
: ПОЗИЦИЯ  ( смещ - смещ+2)  CREATE  DUP ,  2+
         DOES>  ( -- а)  @  УКАЗАТЕЛИ + ;
0  \ начальное смещение
ПОЗИЦИЯ СВЕРХУ
ПОЗИЦИЯ СНИЗУ
ПОЗИЦИЯ СЛЕВА
ПОЗИЦИЯ СПРАВА
ПОЗИЦИЯ ВНУТРИ
ПОЗИЦИЯ СНАРУЖИ
CONSTANT /УКАЗАТЕЛИ  \ конечное вычисленне смещение

CREATE РЕАЛЬНЫЕ  /УКАЗАТЕЛИ ALLOT  \ реальная таблица
CREATE ПСЕВДО  /УКАЗАТЕЛИ ALLOT  \ временная таблица
: РАБОТА       РЕАЛЬНЫЕ 'УКАЗАТЕЛИ ! ;    РАБОТА
: ПРИТВОРСТВО   ПСЕВДО 'УКАЗАТЕЛИ ! ;
```

Слова РАБОТА и ПРИТВОРСТВО соответственно меняют указатели.
К примеру:

```
РАБОТА
10 СВЕРХУ !
СВЕРХУ ? 10

ПРИТВОРСТВО
20 СВЕРХУ !
СВЕРХУ ? 20

РАБОТА
СВЕРХУ ? 10
```

ПРИТВОРСТВО
СВЕРХУ ? 20

Главное отличие в этом последнем подходе заключается в том, что имена проходят через дополнительный уровень перенаправления (УКАЗАТЕЛИ переделаны из константы в определение через двоеточие). Имена полей можно заставить показывать на одну из двух таблиц состояний. Поэтому каждому из них приходится выполнять немного больше работы.

Кроме того, при предыдущем подходе имена соответствовали фиксированным местам в памяти; требовалось применение CMOVE всякий раз, когда мы сохраняли или восстанавливали значения.

При нынешнем подходе нам нужно лишь поменять единственный указатель для смены текущей таблицы.

ВЕКТОРИЗОВАННОЕ ИСПОЛНЕНИЕ

Векторизованное исполнение применяет идеи потоков и перенаправления данных к функциям. Точно так же, как мы сохраняем значения и флаги из переменных, мы можем сохранять и функции, поскольку на последние также можно ссылаться по их адресу.

Традиционная технология введения векторизованного исполнения описана в "Начальном курсе...", в главе 9. В этом разделе мы обсудим придуманный мною новый синтаксис, который, мне кажется, может быть использован во многих случаях более элегантно, чем традиционные методы.

Этот синтаксис называется DOER/MAKE. (Если в Вашей системе отсутствют такие слова, обратитесь к приложению Б, где приведены детали их реализации и кода.) Работает это так: Вы определяете слово, поведение которого будет веторизовываться, с помощью определяющего слова DOER, например:

```
DOER ПЛАТФОРМА
```

Вначале новое слово ПЛАТФОРМА ничего не делает. Затем Вы можете написать слова, которые изменяют то, что делает ПЛАТФОРМА, используя слово MAKE:

```
: ЛЕВОЕ-КРЫЛО   MAKE ПЛАТФОРМА  ." сторонник " ;
: ПРАВОЕ-КРЫЛО  MAKE ПЛАТФОРМА  ." противник " ;
```

Когда вызывается ЛЕВОЕ-КРЫЛО, фраза MAKE ПЛАТФОРМА изменяет то, что должна делать платформа. Теперь, если Вы вводите ПЛАТФОРМА, то получаете:

```
ЛЕВОЕ-КРЫЛО ok

ПЛАТФОРМА сторонник ok
```

ПРАВОЕ-КРЫЛО заставит слово ПЛАТФОРМА печатать "противник".
Можно использовать платформу и внутри другого определения:

```
: ЛОЗУНГ   ." Наш кандидат - последовательный " ПЛАТФОРМА
     ." больших дотаций для промышленности. " ;
```

Выражение

ЛЕВОЕ-КРЫЛО ЛОЗУНГ

будет отражать направление одной предвыборной компании, в то время, как

ПРАВОЕ-КРЫЛО ЛОЗУНГ

будет отражать второе направление.

Код "MAKE"-слова может содержать любые Форт-конструкции, по желанию сколь угодно длинные; следует только помнить о необходимости заканчивать его точкой с запятой. Эта последняя служит в конце левого-крыла не только этому крылу, но и части кода после слова MAKE. Когда слово MAKE перенаправляет слово-DOER, оно одновременно `останавливает` исполнение того определения, в котором само находится.

Когда, к примеру, Вы вызываете ЛЕВОЕ-КРЫЛО, MAKE перенаправляет слово ПЛАТФОРМА и завершает исполнение. Вызов левого-крыла на приводит к распечатке "сторонника". На рисунке 7-7 эта точка показана с использованием иллюстрации состояния словаря.

Рис.7-7. Слова DOER и MAKE.

DOER ДЖО ok

ДЖО

Создает слово DOER с именем ДЖО, которое ничего не делает.

: ТЕСТ MAKE ДЖО 1 . ; ok

ДЖО

ТЕСТ	MAKE	ДЖО	1	.	;

Определяет новое слово с именем ТЕСТ.

ТЕСТ ok

ДЖО

ТЕСТ	MAKE	ДЖО	1	.	;

MAKE перенаправляет ДЖО таким образом, что оно указывает на код после выражения MAKE ДЖО, а затем прекращает исполнение остальной части слова ТЕСТ.

ДЖО 1 ok

Исполняет код, на который указывает ДЖО (1 .).

Если Вы хотите `продолжить` исполнение, то можете использовать вместо точки с запятой слово ;AND. Слово ;AND завершает код, на который направляется слово-DOER и производит исполнение определения, в котором оно применяется, как показано на рис. 7-8.

Наконец, можно связывать `цепи` слов-DOERов при помощи `не` использования слова ;AND. Рисунок 7-9 это поясняет лучше, чем мне удалось бы это описать.

Рис.7-8. Несколько слов MAKE, параллельно использующих ;AND.



Рис.7-9. Последовательное использование нескольких MAKE.

: ТЕСТВ   MAKE ДЖО 4 . MAKE ДЖО 5 . ; <u>**ok**</u>

ТЕСТВ	MAKE	ДЖО	4	.	MAKE	ДЖО	5	.	;

Определение нового слова с именем ТЕСТВ.

ТЕСТВ <u>**ok**</u>

ДЖО → 4

ТЕСТВ	MAKE	ДЖО	4	.	MAKE	ДЖО	5	.	;

MAKE перенаправляет ДЖО на код после MAKE ДЖО.

ДЖО <u>**4 ok**</u>

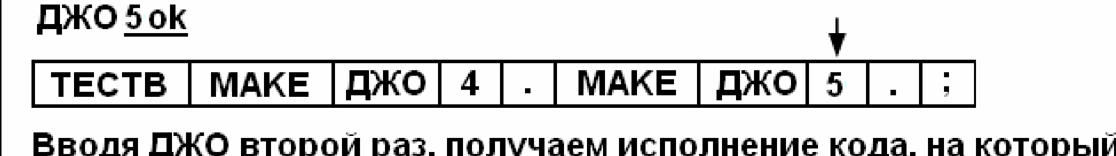
ТЕСТВ	MAKE	ДЖО	4	.	MAKE	ДЖО	5	.	;

Исполнение кода, на который указывает ДЖО (4 . MAKE ...)

ДЖО → 5

ТЕСТВ	MAKE	ДЖО	4	.	MAKE	ДЖО	5	.	;

После исполнения 4 . следующий MAKE перенаправляет ДЖО так, чтобы он показывал на пятерку. (Не было ;AND чтобы его остановить.)



Вводя ДЖО второй раз, получаем исполнение кода, на который он теперь указывает (5 .). Здесь указатель и останется.

ИСПОЛЬЗОВАНИЕ DOER/MAKE

Есть множество случаев, для которых конструкция DOER/MAKE доказала свою полезность. Вот они:

1. Для изменения состояния функции (когда внешняя проверка этого состояния необязательна). Слова ЛЕВОЕ-КРЫЛО и ПРАВОЕ-КРЫЛО изменяют состояние слова ПЛАТФОРМА.
2. Для факторизации внутренних фаз из определений родственных, но лежащих внутри таких структур управления, как циклы.

Вообразите себе определение слова по имени ДАМП, рассчитанного для исследования определенной области памяти.

```
: ДАМП  ( a # )
    0 DO  I 16 MOD 0= IF  CR  DUP I +  5 U.R  2 SPACES  THEN
    DUP I +  @  6 U.R  2 +LOOP  DROP ;
```

Проблема возникает, когда Вы пишете определение с именем СДАМП, расчитанным на печать не в по-словном, а по-байтном формате:

```
: ДАМП  ( a # )
    0 DO  I 16 MOD 0= IF  CR  DUP I +  5 U.R  2 SPACES  THEN
    DUP I +  C@  4 U.R  LOOP  DROP ;
```

Код в этих двух определениях одинаков, за исключением подчеркнутых фрагментов. Но их факторизация затруднена, поскольку они находятся внутри циклов DO LOOP.

Вот решение этой проблемы с использованием DOER/MAKE. Код, содержащий различия, был заменен на слово .ЯЧЕЙКИ, поведение которого векторизуется кодом в ДАМПе и СДАМПе. (Обратите внимание на то, что "1 +LOOP" дает тот же эффект, что и "LOOP".)

```
DOER .ЯЧЕЙКИ  ( a -- прибавка ) \ распечатать байт или слово
    : <ДАМП>  ( a # )
        0 DO  I 16 MOD 0= IF  CR  DUP I +  5 U.R  2 SPACES  THEN
        DUP I +  .ЯЧЕЙКИ  +LOOP  DROP ;
    : ДАМП   ( a # )  MAKE .ЯЧЕЙКИ  @  6 U.R  2 ;AND  <ДАМП> ;
    : СДАМП  ( a # )  MAKE .ЯЧЕЙКИ C@  4 U.R  1 ;AND  <ДАМП> ;
```

Обратите внимание на то, как ДАМП и СДАМП `выставляют` вектор, а затем переходят к `исполнению` (слово <ДАМП>).

3. Для изменения состояния родственных функций после вызова единственной команды. К примеру:

```
DOER TYPE'
DOER EMIT'
DOER SPACES'
DOER CR'
: ВИДИМО   MAKE TYPE'        TYPE ;AND
           MAKE EMIT'        EMIT ;AND
           MAKE SPACES'      SPACES ;AND
           MAKE CR' CR ;
: НЕВИДИМО        MAKE TYPE'        2DROP ;AND
                  MAKE EMIT'        DROP ;AND
                  MAKE SPACES'      DROP ;AND
                  MAKE CR' ;
```

Мы здесь определили набор векторизованных определений для вывода, имена которых имеют на конце значок "вторичности".

Слово ВИДИМО устанавливает их на соответствующие функции. Слово НЕВИДИМО переводит их в нерабочее положение, съедая аргументы, которые

нормально должны были бы быть ими использованы. Говорим НЕВИДИМО - и все слова, определенные в терминах этих четырех операторов вывода `не` будут производить вывод.
4. Для изменения состояния только для одного следующего вхождения, а затем изменение состояния (или восстановления) вновь. Представим себе, что мы пишем приключенческую игру. Когда игрок впервые входит в определенную комнату, игра должна показать ее подробное описание. Если же он позже вновь в нее позже возвращается, игрушка должна выдать короткое сообщение. Мы пишем:

```
DOER АНОНС
: ДЛИННЫЙ   MAKE АНОНС
     CR ." Вы в большом тронном зале с высоким троном,"
     CR ." покрытым красным бархатным ковром."
MAKE АНОНС
     CR ." Вы в тронном зале." ;
```

Слово АНОНС будет показывать одно из сообщений. Вначале мы говорим ДЛИННЫЙ, инициализируя АНОНС на длинное сообщение. Теперь мы можем проверить АНОНС и убедиться, что он действительно печатает подробное описание. После завершения этого следующим этапом он "делает" анонс коротким. Если мы снова попробуем АНОНС, то он напечатает лаконичное сообщение. И так будет до тех пор, пока мы не скажем ДЛИННЫЙ опять.

В результате мы устанавливаем очередь поведений. Мы можем создать такую очередь для любого числа поведений, позволяя каждому из них выставлять последующее. Нижеприведенный пример (хотя и не ужасно практически полезный) иллюстрирует такой метод.

```
DOER ГДЕ
VARIABLE РУБАШКА
VARIABLE ШТАНЫ
VARIABLE ГАРДЕРОБ
VARIABLE МАШИНА

: ПОРЯДОК  \  определить порядок поиска
     MAKE ГДЕ  РУБАШКА  MAKE ГДЕ  ШТАНЫ
     MAKE ГДЕ ГАРДЕРОБ  MAKE ГДЕ  МАШИНА
     MAKE ГДЕ  0 ;

: ШАРИТЬ  ( -- a|0 )  \ искать место, где находится 17
     ПОРЯДОК  5 0 DO  ГДЕ  DUP 0=  OVER @  17 =  OR  IF
        LEAVE  ELSE  DROP  THEN  LOOP ;
```

В этом отрывке мы создали список переменных, затем определили ПОРЯДОК, в котором они должны просматриваться. Слово ШАРИТЬ проверяет каждую из них в поисках содержащей число 17. ШАРИТЬ возвращает либо адрес соответствующей переменной, либо ноль, если ни одна из них не содержит этого числа.

Оно делает это, просто исполняя слово ГДЕ пять раз. И каждый раз ГДЕ возвращает разные адреса, по порядку, и в конце концов - ноль. Мы можем даже определить слово-DOER, которое бесконечно переключает свое собственное поведение:

```
DOER РЕЧЬ
: ПЕРЕКЛЮЧИТЬ
```

```
     BEGIN  MAKE РЕЧЬ  ." ПРИВЕТ "
MAKE РЕЧЬ  ." ПРОЩАЙ "
     0 UNTIL ;
```

5. Для создания ссылки вперед. Такая ссылка обычно нужна в качестве "затычки", то есть слова, вызываемого в определении нижнего уровня, действие которого определится только в компоненте, который будет создан в программе позже.
Для реализации ссылки вперед постройте слово с помощью DOER до первого использования его имени.

```
DOER ПОКА-НЕ-ОПРЕДЕЛЕНО
```

Позже в программе используйте MAKE:

```
MAKE ПОКА-НЕ-ОПРЕДЕЛЕНО  ВЕСЬ ЭТОТ ШУМ ;
```

(Помните, что MAKE может использоваться и вне определения через двоеточие.)
6. Рекурсия, прямая и косвенная. Прямая рекурсия возникает тогда, когда слово обращается к самому себе. Хорошим примером может послужить рекурсивное определение наибольшего-общего-делителя:

```
НОД от а, б =          а                            если б = 0
                 НОД от б, а mod б            если б > 0
```

Это отлично переводится в:

```
DOER НОД  ( а б -- нод)
MAKE НОД  ?DUP  IF  DUP ROT ROT  MOD  НОД  THEN ;
```

Косвенная рекурсия имеет место тогда, когда одно слово вызывает второе слово, а второе слово вызывает первое. Это можно устроить, используя конструкцию:

```
DOER Б
: А   ... Б ... ;
MAKE Б   ... А ... ;
```

7. Отладка. Я часто определяю:

```
DOER SNAP
```

(сокращение от SNAPSHOT - моментальный снимок), затем внедряю SNAP в свою задачу в точке, в которой хочу ее проконтролировать. К примеру, вызывая SNAP внутри цикла интерпретатора клавиатуры, я могу так настроить SNAP, чтобы наблюдать за тем, что происходит в структуре данных по мере нажатия клавиш. И я могу изменить функцию SNAP без перекомпиляции всего цикла.

Ситуации, в которых предпочтителен подход с использованием ' и EXECUTE, возникают тогда, когда необходимо передать управление через адрес вектора, например, при векторизации через таблицу решений, или при сохранении/восстановлении содержимого вектора.

ИТОГИ

В этой главе мы исследовали все за и против использования стека или переменных и других структур данных. Использование стека предпочтительней для обеспечения тестирования и возможности повторного использования, однако слишком большое количество чисел для манипуляций на стеке в пределах одного определения вредит удобству как при написании, так и при чтении программ.

Мы исследовали также технику сохранения и восстановления структур данных и в заключение изучили векторизованное исполнение с использованием конструкций DOER/MAKE.

ЛИТЕРАТУРА

1. Michael Ham, "Why Novices Use So Many Variables," `FORTH Dimensions`, vol. 5, no. 4, November/December 1983.
2. Daniel Slater, "A State Space Approach to Robotics," `The Journal of FORTH Applications and Research`, 1, 1 (September 1983), 17.

ГЛАВА 8

МИНИМИЗАЦИЯ

СТРУКТУР УПРАВЛЕНИЯ

Структуры управления в Форте играют меньшую роль, нежели в других языках. Форт-программисты стараются описывать очень сложные приложения в терминах коротких слов и без особого акцента на конструкции IF THEN.

Имеется несколько приемов для минимизации структур управления. Они включают в себя:

- вычисления или подсчет
- упрятывание условных операторов при рефакторизации
- использование структурированных выходов
- перепроектирование.

В этой главе мы исследуем эти приемы с целью упрощения и удаления структур управления из Вашего кода.

ЧТО ЖЕ ТАКОГО ПЛОХОГО В СТРУКТУРАХ УПРАВЛЕНИЯ?

Перед тем, как мы станем разворачивать свой список советов, давайте приостановимся для того, чтобы понять, зачем следует в первую очередь избегать условных операторов.

Использование структур условного управления добавляет сложность в Ваш код. Чем он сложнее, тем сложнее Вам будет его читать и поддерживать. Чем из большего числа деталей состоит машина, тем больше шансов у нее сломаться. И тем труднее будет ее чинить.

Мур рассказывает историю:

> Недавно я вновь связывался с компанией, для которой мы сделали некую работу несколько лет назад. Они меня вызвали потому, что их программе теперь стукнуло пять лет и она стала очень сложной. Их программисты влезали туда и многое меняли, добавляя переменные состояния и операторы условных переходов. Каждое выражение, которое я держал за простое пять лет назад, теперь стало очень сложным. "Если это, иначе если то, иначе если се" ...и уже только потом простое выражение.
>
> Теперь, читая эти фразы, я не могу выделить, что же происходит и почему. Я должен был бы помнить, что означает каждая из переменных, почему она применена именно в этом случае и затем: что происходит, когда данная последовательность происходит - или не происходит.

Начиналось все самым невинным образом. У них был особый случай, о котором надо было позаботиться. Для обеспечения этого они ввели условный оператор в одном месте. Затем они обнаружили, что такой же нужен здесь и здесь. А затем — еще несколько. Каждый шаг по возрастающей лишь добавлял небольшой сумбур в программу. Будучи программистами, они каждый раз превосходили это.

Общий эффект был разрушительным. В конце концов у них образовалось с полдюжины флагов. Проверить этот, сбросить его, установить тот и т.д. В качестве результата одного условия нужно было проверять следующие возникающие условия.

Они создали логический эквивалент макаронообразного кода вместо того, чтобы использовать возможности структурирования программы. Сложность зашла гораздо дальше того уровня, который они предполагали. Однако они обрекли себя на этот путь, пропустив простое решение, которое все это могло бы сделать ненужным - иметь два слова вместо одного. Вы говорите либо РАБОТАТЬ, либо ДЕЛАТЬ-ВИД.

В большинстве приложений имеется очень немного мест, в которых Вам нужно проверять условие. К примеру, в видеоигре Вы в действительности не говорите: "Если нажимают Кнопку А, то сделать это; если нажимают Кнопку Б, то сделать что-нибудь другое". Вы с подобной логикой не связываетесь. Если кнопку нажимают, то Вы что-то исполняете. Это что-то связано именно с кнопкой, а не с логикой.

Условные операторы сами по себе не плохи - это конструкции нужные. Но программа с большим количеством таких конструкций запутана и нечитаема. Все, что Вам нужно делать - это по каждой из них задать себе вопрос. Любой условный оператор должен заставить Вас спросить: "Что я делаю неправильно?"

С условными операторами можно пытаться совладать различными способами. Длинные вложенные последовательности иного рода лучше длинных вложенных последовательностей условных переходов.

Перед тем, как мы предложим некоторые конкретные приемы, давайте взглянем на три подхода к использованию условных операторов на конкретном примере. Рисунки 8-1, 8-2 и 8-3 иллюстрируют три версии проекта автоматической банковской машины.

Первый пример вышел прямо из Школы Структурированных Программистов. Логика задачи зависит от правильной вложенности операторов IF.

Рис.8-1. Структурированный подход.

```
        БАНКОМАТ

IF  карточка правильная  DO
        IF  владелец карточки правильный  DO
                IF  запрос на оплату  DO
                        IF  код пароля правильный  DO
                                запрос количества
                                IF  запрос =< текущего баланса  DO
                                        IF  расход =< допустимого расхода  DO
                                                оплатить запрос
                                                подсчитать дебет
                                        ELSE
```

```
                        предупреждение
                        конец сеанса
                ELSE
                    предупреждение
                    конец сеанса
            ELSE
                предупреждение
                конец сеанса
        ELSE
            IF  код пароля правильный  DO
                запрос количества
                получить конверт через щель
                подсчитать кредит
            ELSE
                предупреждение
                конец сеанса
    ELSE
        съесть карточку
ELSE
    предупреждение
КОНЕЦ
```

Легко читается? Скажите мне, при каком условии пользовательская карточка будет съедена. Для ответа Вам придется либо подсчитать ELSEы от конца и приложить их к тому же количеству IFов от начала, либо использовать линейку.

Во второй версии на рис. 8-2 показано улучшение за счет использования множества мелких именованных процедур для удобочитаемости. Карточка пользователя будет съедена, если владелец не тот.

Рис.8-2. Вложенные условные операторы внутри именованных процедур.

```
    БАНКОМАТ

PROCEDURE ЧИТАТЬ-КАРТУ
    IF  карточка читается  THEN  ПРОВЕРИТЬ-ВЛАДЕЛЬЦА
        ELSE  выбросить карточку  КОНЕЦ

PROCEDURE ПРОВЕРИТЬ-ВЛАДЕЛЬЦА
    IF  владелец правильный  THEN  ПРОВЕРИТЬ-КОД
        ELSE  съесть карточку  КОНЕЦ

PROCEDURE ПРОВЕРИТЬ-КОД
    IF  введенный код соответствует владельцу  THEN  ТРАНЗАКТ
        ELSE  предупреждение, конец сеанса  КОНЕЦ

PROCEDURE ТРАНЗАКТ
    IF  запрос на оплату  THEN  РАСХОД
        ELSE  ПРИХОД  КОНЕЦ

PROCEDURE РАСХОД
    запрос
    IF  запрос =< текущий баланс  THEN  ОПЛАТИТЬ  КОНЕЦ
```

```
PROCEDURE ОПЛАТИТЬ
        IF  сумма =< допустимый расход  THEN
                оплатить счет
                подсчитать дебет
        ELSE  предупреждение  КОНЕЦ

PROCEDURE ПРИХОД
        принять конверт
        подсчитать кредит
```

Однако даже после такого улучшения структура каждого слова полностью зависит от `последовательности`, в которой производятся проверки. Процедура предположительно "высшего" уровня обременена проверкой на наихудший, самый тривиальный тип события. И каждая из проверок становится ответственной за вызов следующей.

Третья версия ближе всего подходит к тому, что может дать Форт. Слово самого высокого уровня точно выражает, что происходит на уровне концептуальном, показывая лишь главный путь. Каждое их подчиненных слов имеет свой собственный выход по ошибке, не засоряющий читабельность главного слова. Одна проверка не обязана вызывать следующую проверку.

Рис.8-3. Перефакторизация и/или устранение условных операторов.

```
        БАНКОМАТ

: ПУСК
        ЧИТАТЬ-КАРТУ  ТЕСТ-ВЛАДЕЛЬЦА  ТЕСТ-КОДА  ТРАНЗАКТ ;

: ЧИТАТЬ-КАРТУ
        правильная послед-ть кодов НЕ читается
         IF  выбросить карточку  QUIT  THEN ;

: ТЕСТ-ВЛАДЕЛЬЦА
        владелец НЕ правильный IF  съесть карточку QUIT  THEN ;

: ТЕСТ-КОДА
        введенный код НЕ соответствует коду владельца
         IF  предупреждение  QUIT  THEN ;

: ЧИТАТЬ-КНОПКУ  ( -- адрес-функции-кнопки )
        ( аппаратно-зависимый примитив ) ;

: ТРАНЗАКТ
        ЧИТАТЬ-КНОПКУ EXECUTE ;

1 КНОПКА РАСХОД

2 КНОПКА ПРИХОД

: РАСХОД
        запрос
```

```
        запрос =< текущего баланса  IF  ОПЛАТА  THEN ;
: ОПЛАТА
        оплата =< допустимого расхода  IF
                оплатить
                подсчитать дебет
        ELSE  предупреждение  THEN ;

: ПРИХОД
        принять конверт
        подсчитать кредит ;
```

Слово ТРАНЗАКТ, кроме того, спроектировано вокруг того факта, что пользователь делает запросы, нажимая кнопки на клавиатуре. Нет необходимости ни в каких условных переходах.

Одна кнопка начинает запрос на оплату, а другая — на размещение. Этот подход готов к восприятию позднейших изменений, таких, как добавление возможности делать переводы средств. (И этот подход поэтому становится `не` зависимым от аппаратуры. Детали интерфейса с клавиатурой могут быть сокрыты в лексиконе клавиатуры ЧИТАТЬ-КНОПКУ и КНОПКА.)

Разумеется, Форт позволит Вам выбрать любой из трех методов. Который из них Вы предпочитаете?

КАК УСТРАНЯТЬ СТРУКТУРЫ УПРАВЛЕНИЯ

В этом разделе мы изучим различные способы упрощения или избегания условных операторов. Большинство из них будет давать более читабельный, лучше приспособленный к сопровождению и более эффективный код. Некоторые из приемов дают более эффективный, но не всегда столь же хорошо читабельный код.

Помните поэтому: Не все советы применимы во всех ситуациях.

ИСПОЛЬЗОВАНИЕ СЛОВАРЯ.

СОВЕТ
Давайте всякой функции свое определение.

Правильным образом используя словарь Форта, мы в действительности не сокращаем число условных операторов; мы просто вычленяем их из кода нашего приложения. Словарь Форта - это гигантский оператор CASE со строковыми аргументами. Функции сравнения и исполнения скрыты внутри Форт-системы.

Мур:

В моем приходно-расходном пакете, когда Вы получаете от кого-нибудь чек, то пишете сумму, номер чека, слово ОТ и имя этого кого-то:

```
200.00 127 ОТ АЛИ-БАБА
```

Слово ОТ заботится о данной ситуации. Если Вы хотите выставить кому-нибудь счет, Вы пишете сумму, номер вызова, слово СЧЕТ и имя:

1000.00 280 СЧЕТ ТЕХНИТЕКС

... По одному слову на каждый случай. Решения принимает словарь.

Это замечание характеризует весь Форт. Для сложения пары чисел одинарной длины мы используем команду +. Для сложения пары чисел двойной длины - слово D+. Менее эффективный, более сложный подход использовал бы единственную команду, которая как-то "разбирала" бы, какие из типов чисел должны быть сложены.

Форт эффективен потому, что все эти слова - ОТ и СЧЕТ и + и D+ - могут быть реализованы безо всякой необходимости в проверках и условных переходах.

СОВЕТ
Используйте "тупые" слова.

Это вовсе не рекомендация для телевизионных сценаристов. Это - еще одна ипостась использования словаря. "Тупое" слово — это такое слово, которое не зависит от ситуации, но имеет одно и то же действие все время ("очевидно при ссылках").

В тупом слове нет неопределенности, а потому оно заслуживает большего доверия.

Несколько обычных слов в Форте к выходу этой книги являлись источником противостояний. Одним из таких слов является .", которое печатает строку. В простейшем виде оно допустимо только внутри определения через двоеточие:

```
: ПРОВЕРКА   ." ЭТО - СТРОКА. " ;
```

В действительности эта версия слова \`не\` печатает строку. Она \`компилирует\` строку вместе с адресом другого определения, которое ее и печатает во время исполнения.

Это тупая версия слова. Если Вы используете ее вне определения через двоеточие, оно бессмысленно скомпилирует строку, что совсем не соответствует тому, что ожидает получить новичок.

Для решения этой проблемы FIG-Форт добавил проверку внутрь слова .", которая определяла, находится ли система в режиме компиляции или интерпретации. В первом случае оно компилировало строку и адрес примитива, во втром - печатало ее TYPEом.

Слово ." стало состоящим из двух совершенно разных слов, поселенных вместе в одном определении внутри структуры IF THEN ELSE. Флаг, определяющий, компилирует или интерпретирует Форт, называется STATE. Поскольку ." зависит от STATE, то говорят, что это слово "STATE-зависимо".

Команда \`вела себя\` одинаково внутри и снаружи определения через двоеточие. Эта двойственность оправдала себя для послеобеденных вводных курсов по Форту, но серьезный студент скоро понял, что это еще не все.

Допустим, некая студентка желает написать новое слово по имени Я." (т.е. "яркая-точка-кавычка") для отображения выделенной яркостью строки символов на своем дисплее, с использованием в виде:

." Вставьте диск в " Я." левый " ." привод "

Она могла бы расчитывать определить это так:

: Я."   ЯРКО ." НОРМА ;

то есть поменять видеорежим на повышенную яркость, напечатать строку, а затем восстановить режим на нормальный.

Она пытается. И немедленно - иллюзия разрушена; обман разоблачен; определение не может работать.

Для решения ее проблемы программистке придется изучить определение слова (.") в своей собственной системе. Я не собираюсь здесь производить отступления по поводу того, как работает (.") - на мой взгляд, красотой оно не выделяется.

Между прочим, имеется и другой синтаксический подход к проблеме нашей студентки, а именно тот, который не требует наличия двух раздельных слов ." и Я." для печати строк. Измените системное слово (.") так, чтобы оно всегда устанавливало нормальный режим после печати, даже если он и так будет нормальным большую часть времени. Имея такой синтаксис, программистке нужно будет просто предварить выделяемую строку

простым словом ЯРКО.

." Вставьте диск в "   ЯРКО ." левый "   ." привод "

Стандарт-83 ныне специфицирует тупое определение .", а для случаев, когда необходима интерпретирующая версия добавлено новое слово .(. К счастью, в этом стандарте мы используем словарь для принятия решения с помощью двух раздельных слов.

Слово ' (штрих) имеет подобную же историю. В Фиг-Форте оно было STATE-зависимым, но теперь в Стандарте-83 оно тупое. И штрих, и точка-кавычка характерны тем, что программист может захотеть повторно использовать одно из этих слов в высокоуровневом определении и при этом ожидать, что они будут вести себя так же, как и в обычных условиях.

СОВЕТ

Слова не должны зависеть от переменной STATE, если программист собирается когда-либо использовать их для вызова из высокоуровневого определения и при этом ожидает получить от них такое же поведение, как и при интерпретации.

В качестве STATE-зависимого определения хорошо работает слово ASCII, равно как и слово MAKE. (См. приложение B.)

ВЛОЖЕННОСТЬ И КОМБИНАЦИИ УСЛОВНЫХ ОПЕРАТОРОВ.

СОВЕТ

Не делайте проверок на то, что уже было исключено.

Пожалуйста, возьмите такой пример:

```
: ОБРАБОТАТЬ-КЛАВИШУ
        KEY  DUP  СТРЕЛКА-ВЛЕВО  =  IF  КУРСОР-ВЛЕВО   THEN
        DUP  СТРЕЛКА-ВПРАВО =  IF  КУРСОР-ВПРАВО  THEN
        DUP  СТРЕЛКА-ВВЕРХ  =  IF  КУРСОР-ВВЕРХ   THEN
        СТРЕЛКА-ВНИЗ   =  IF  КУРСОР-ВНИЗ    THEN ;
```

Эта версия неэффективна, поскольку все проверки должны делаться независимо от исхода любой из них. Если была бы нажата, скажем, клавиша-влево, то не было бы необходимости в проверке ее на другие совпадения.

Вместо этого можно было бы сделать вложенные проверки, типа:

```
: ОБРАБОТАТЬ-КЛАВИШУ
        KEY  DUP  СТРЕЛКА-ВЛЕВО  =  IF  КУРСОР-ВЛЕВО   ELSE
        DUP  СТРЕЛКА-ВПРАВО =  IF  КУРСОР-ВПРАВО  ELSE
        DUP  СТРЕЛКА-ВВЕРХ  =  IF  КУРСОР-ВВЕРХ   ELSE
КУРСОР-ВНИЗ
        THEN THEN THEN  DROP ;
```

СОВЕТ

Компануйте вместе булевские значения одинакового веса.

Многие структуры IF THEN двойной вложенности могут быть упрощены с помощью комбинаций флагов и логических операторов перед принятием решения. Вот тестирование с двойной вложенностью:

```
: ?ГУЛЯТЬ   СУББОТА? IF  ДЕЛО СДЕЛАНО? IF
        СМЕЛО ПОЙТИ ГУЛЯТЬ  THEN THEN ;
```

Вышеприведенный код использует вложенные IFы для того, чтобы удостовериться в том, что одновременно и суббота наступила, и дело закончено перед гулянием. Вместо этого давайте скомбиниркем условия логически и произведем единственную проверку:

```
: ?ГУЛЯТЬ   СУББОТА?  ДЕЛО СДЕЛАНО?  AND  IF
        СМЕЛО ПОЙТИ ГУЛЯТЬ  THEN ;
```

Это проще и лучше читается.

При ситуации логического "или" реализация через IF THENы еще ухабистей:

```
: ?ВСТАВАТЬ    ТЕЛЕФОН ЗВОНИТ? IF  ВСТАТЬ  THEN
        БУДИЛЬНИК ЗВОНИТ? IF  ВСТАТЬ  THEN ;
```

Это куда более элегантно записывается в виде

```
: ?ВСТАВАТЬ    ТЕЛЕФОН ЗВОНИТ?  БУДИЛЬНИК ЗВОНИТ?  OR
        IF  ВСТАТЬ  THEN ;
```

Из этого правила имеется одно исключение - а именно тогда, когда слишком велик проигрыш в скорости при выполнении проверок.

Мы могли бы написать:

```
: ?ПОХЛЕБКА   СТРУЧКИ-БОБОВ? ПОХЛЕБКА РЕЦЕПТ?  AND IF
          ПОХЛЕБКА ПРИГОТОВИТЬ  THEN ;
```

Однако предположим, что у нас может занять много времени нашаривание в нашем рецептурном файле чего-нибудь насчет похлебок. Очевидно, что нет смысла предпринимать поиск, если у нас бобов в запасе нет. Было бы более эффективно написать

```
: ?ПОХЛЕБКА   СТРУЧКИ-БОБОВ? IF  ПОХЛЕБКА РЕЦЕПТ?  IF
          ПОХЛЕБКА ПРИГОТОВИТЬ  THEN   THEN ;
```

Мы не обеспокоены поисками рецептов при отсутствии бобов.

Другое исключение возникает, когда какое-нибудь условие не выполняется скорее всего. Устраняя его в первую очередь, Вы избегаете необходимости проверять остальные.

СОВЕТ

Когда многочисленные условия имеют различный вес (в смысле предпочтительности или времени на вычисление), вкладывайте условные операторы так, чтобы на первом уровне был тот, который выполняется наименее вероятно или является самым легким в вычислении.

Сложнее пытаться улучшить подобным образом конструкции с OR. К примеру, в определении

```
: ?ВСТАВАТЬ    ТЕЛЕФОН ЗВОНИТ?  БУДИЛЬНИК ЗВОНИТ?  OR
          IF  ВСТАТЬ  THEN ;
```

мы проверяем и телефон, и будильник, хотя лишь одного из них достаточно, чтобы нас поднять. Теперь предположим, что определять, что звонит будильник, очень трудоемко. Мы могли бы написать

```
: ?ВСТАВАТЬ    ТЕЛЕФОН ЗВОНИТ? IF  ВСТАТЬ  ELSE
          БУДИЛЬНИК ЗВОНИТ?  IF  ВСТАТЬ  THEN  THEN ;
```

Если верно первое условие, нам больше не нужно проверять второе. Мы в любом случае вынуждены вставать, чтобы взять трубку. Повторение ВСТАТЬ выглядит уродливо и даже близко не подходит по читабельности решению через OR - однако в некоторых случаях оно предпочтительно.

ВЫБОР СТРУКТУР УПРАВЛЕНИЯ

СОВЕТ

Самый элегантный код - тот, который наиболее точно отражает проблему. Выбирайте структуру управления так, чтобы она наилучшим образом подходила к проблеме передачи управления.

ОПЕРАТОРЫ CASE.

Определенный класс задач требует выбора нескольких возможных путей исполнения в соответстсвии с числовым аргументом. К примеру, мы желаем написать слово .МАСТЬ для приема числа, представляющего масть игральной карты, от 0 до 3, и печати имени этой масти. Можно было бы написать это слово с использованием вложенных операторов IF ELSE THEN, типа:

```
: .МАСТЬ  ( масть -- )
        DUP   0= IF    ." ЧЕРВИ "    ELSE
        DUP  1 = IF    ." ПИКИ "     ELSE
        DUP  2 = IF    ." КРЕСТИ "  ELSE
                       ." БУБИ "
        THEN THEN THEN  DROP ;
```

Мы можем решить эту задачу элегантнее при использовании "оператора CASE". Вот то же самое определение, переписанное с использованием формата "Экеровсокого оператора CASE", названного так по имени Др. Чарльза Э. Экера, джентльмена, его предложившего [1].

```
: .МАСТЬ  ( масть -- )
CASE
0 OF  ." ЧЕРВИ "      ENDOF
1 OF  ." ПИКИ "       ENDOF
2 OF  ." КРЕСТИ "   ENDOF
3 OF  ." БУБИ "       ENDOF    ENDCASE ;
```

Достоинство оператора CASE состоит исключительно в удобстве его чтения и написания. Он не дает улучшения эффективности ни по занимаемой памяти, ни по скорости исполнения. На самом деле такой оператор компилирует во многом тот же код, что и вложенные операторы IF THEN. Оператор CASE - хороший пример факторизации во время компиляции.

Должна ли всякая Форт-система включать в себя этот оператор? Это - палка о двух концах. Во-первых, случаи, когда такая конструкция действительно нужна, редки – достаточно редки, чтобы поставить под вопрос ее ценность. Если встречаются всего несколько таких случаев, конструкция IF ELSE THEN тоже будет неплохо работать, хотя и, быть может, при худшей читабельности. Если таких случаев много, более гибкой является таблица решений.

Во-вторых, многие задачи подобного типа не совсем подходят для структуры CASE. Экеровский оператор предполагает, что Вы проверяете на равенство число на стеке. В примере со словом .МАСТЬ у нас имеется непрерывный ряд чисел от 0 до 3. Эффективней было бы использовать целое для вычисления смещения и непосредственного перехода на нужный код.

В случае Крошечного Редактора (позже в этой главе) у нас имеются не одно, а два измерения возможностей. Оператор CASE для такой проблемы тоже не подходит.

Лично я считаю оператор CASE элегантным решением для неправильно поставленной задачи: попыткой алгоритмического выражения того, что более точно описывается таблицей решений.

Оператору CASE следует быть частью задачи тогда, когда он бывает полезен, но не частью системы.

ЦИКЛИЧЕСКИЕ СТРУКТУРЫ.

Правильно выбранная структура цикла может устранить дополнительные условные операторы.

Мур:

Во многих случаях условные структуры используются для выхода из циклов. Эту конкретную ситуацию можно избежать, строя циклы с многочисленными точками выхода. Это тема весьма жизненна, поскольку множественные конструкции WHILE имеются в polyFORTHe, хотя и не дошли до Форта '83. Они являются простым способом определения многих WHILEов при одном REPEATе.

Также и Дин Сендерсон (из фирмы FORTH, Inc.) изобрел новую конструкцию, которая дает две точки выхода для цикла DO LOOP. Имея такую конструкцию, Вы делаете меньше проверок. Очень часто я держу на стеке значение "истина", а если покидаю цикл рано, то меняю это значение, чтобы знать, что я покинул его рано. Затем позже я использую IF для проверки того, когда я покинул цикл, и это очень неуклюже.

Если однажды решение принято, не следует принимать его опять. Имея подходящие конструкции для циклов, Вам не придется помнить, откуда Вы пришли, и большее количество проверок будет устранено.

Это не совсем популярно, потому что слегка неструктурированно. Или, может быть, чрезмерно структурировано. Ценно то, что программы получаются проще. И это ничего не стоит.

Разумеется, проблема эта жизненна. Здесь, наверное, слишком рано было бы предлагать какие-то определенные новые конструкции циклов. Проверьте документацию на Вашу систему на предмет того, что в ней предлагается из экзотических цикловых структур. Или, по нуждам Ваших задач, добавьте свои собственные структуры. Это не так уж сложно в Форте.

Я даже не вполне уверен в том, насколько использование множественных выходов не противоречит доктринам структурированного программирования. В цикле BEGIN WHILE REPEAT с многими WHILEами все выходы приводят к одной точке продолжения: REPEAT. Однако в конструкции Сендерсона можно выходить из цикла, перепрыгивая \`через\` конец цикла и продолжая исполнение после ELSE. Имеются две возможные точки продолжения.

Это "хуже структурировано", если можно так выразиться. И в то же время определение всегда закончится точкой с запятой и вернет управление вызвавшему его слову. В этом смысле оно хорошо структурировано; модуль имеет одну точку входа и одну точку выхода.

Когда Вы хотите выполнить некий код только если \`не\` покинули цикл преждевременно, использовать такой подход кажется самым естественным. (Мы

рассмотрим пример этому в последующем разделе "Использование структурированных выходов".)

СОВЕТ
Предпочитайте счетчики перед проверками на окончание.

Форт поддерживает формат строк, содержащих длину в первом байте. Это упрощает печать, перемещение и вообще любые действия со строками. При адресе и счетчике, лежащих на стеке, определение слова TYPE может быть таким:

```
: TYPE  ( a #)  OVER + SWAP DO  I C@ EMIT  LOOP ;
```

(Хотя TYPE в действительности обязано быть написанным в машинных кодах.)

Это определение не использует впрямую условного оператора. LOOP на самом деле скрывает в себе такой оператор, поскольку каждый цикл проверяет индекс и делает возврат на DO, если он еще не достиг предела.

Если бы использовались символы-ограничители, скажем, ASCII нуль, то определение должно было бы быть написано как:

```
: TYPE  ( a)  BEGIN  DUP C@  ?DUP WHILE  EMIT 1+
          REPEAT  DROP ;
```

Нужна дополнительная проверка при каждом прохождении цикла. (WHILE является оператором условного перехода.)

Замечание по оптимизации: использование слова ?DUP с точки зрения времени исполнения накладно, поскольку оно само содержит дополнительную проверку. Более быстрым было бы определение:

```
: TYPE  ( a)  BEGIN  DUP C@  DUP WHILE  EMIT 1+
          REPEAT  2DROP ;
```

Стандарт '83 применил этот принцип в слове INTERPRET, которое ныне воспринимает счетчик вместо поиска символа-ограничителя.

Оборотной стороной этой медали являются некоторые структуры данных, которые легче `связывать` вместе. Каждая запись указывает на последующую (или предыдущую). Последняя (или первая) запись в цепи может обозначаться нулем в своем поле связи.

Раз у Вас есть поле связи, Вам все равно придется работать с его значением. Нет необходимости хранить счетчик того, сколько имеется записей. Если Вы всякий раз уменьшаете счетчик для выяснения момента окончания, то создаете себе этим лишнюю работу. (Такая технология применяется при создании словаря Форта как связного списка.)

ВЫЧИСЛЕНИЕ РЕЗУЛЬТАТОВ.

СОВЕТ
Не принимайте решений, но вычисляйте.

Во многих случаях условные операторы применяются ошибочно тогда, когда различные варианты поведения обусловливаются численными различиями. Раз задействованы числа, то мы их можем посчитать. (См. в главе 4 раздел "Расчеты или структуры данных или логика".)

СОВЕТ
Используйте логические значения в качестве чисел.

Это высказывание замечательно венчает собой предыдущий совет "Не принимайте решений, но вычисляйте". Идея состоит в том, что булевские значения, которые в компьютере представлены числами, могут быть эффективно использованы для улучшения принятия решений. Вот один пример, встречающийся во многих Форт-системах:

```
: S>D  ( n -- d ) \ распространение знака на d
         DUP 0< IF  -1  ELSE  0  THEN ;
```

(Задачей этого определения является превращение числа одинарной длины в число двойной длины. Это последнее представляется как два 16-разрядных числа на стеке, старшие разряды - на вершине. Превращение положительного целого заключается просто в добавлении нуля на стек для представления этой старшей части. Но преобразование отрицательного требует "расширения знака", то есть старшая часть должна быть представлена всеми единицами.)

Вышеприведенное определение проверяет одинарное число на отрицательность. Если таковая обнаружена, оно кладет на стек минус-единицу; в противном случае - ноль.

Но обратите внимание, что выход получается чисто арифметическим; качественных изменений самого процесса нет. Из этого факта можно извлечь выгоду, используя само логическое
значение:

```
: S>D  ( n -- d ) \ распространение знака на d
         DUP 0< ;
```

Эта версия оставляет на стеке ноль или минус-единицу без принятия решений.
(В системах до 83-го стандарта определение должно было бы быть таким):

```
: S>D  ( n -- d ) \ распространение знака на d
         DUP 0< NEGATE ;
```

См. приложение В.)
С такими "гибридными числами" можно проделывать даже больше:

СОВЕТ
Для повышения эффективности формирования числового выхода
используйте AND.

Для случая структуры, принимающей решение о формировании нуля либо ненулевого числа "n", традиционная фраза

(?) IF n ELSE 0 THEN

эквивалентна более простому выражению

(?) n AND

Опять же секрет в том, что в системах стандарта '83 "истина" представляется -1 (всеми единичками). Делая AND числу "n" с флагом, мы оставляем либо "n" (все биты остаются), либо "0" (все биты очищаются).

Для закрепления - еще пример:

(?) IF 200 ELSE 0 THEN

то же самое, что

(?) 200 AND

Посмотрите на следующий случай:

n a b < IF 45 + THEN

Фраза либо добавляет 45 к "n", либо нет, в зависимости от соотношения величин "a" и "b". Поскольку "добавить 45 или нет" - это то же, что и "добавить 45 или добавить 0", то различие в двух возможных исходах - чисто числовое. Мы можем позволить себе избежать принятия решения и просто вычислить:

n a b < 45 AND +

Мур:

> Выражение "45 AND" работает быстрее, чем IF, и, конечно, более грациозно. Оно проще. Если Вы освоите привычку обращать внимание на точки, в которых вычисляете отличие одного значения от другого, то обычно при выполнении арифметических действий над логическими величинами Вы будете получать тот же результат более чистым образом. Я не знаю, как это называется. Здесь нет терминологии - просто выполнение арифметики со значениями "истина". Но это совершенно корректно, и когда-нибудь булева алгебра и арифметические выражения к этому приспособятся.
>
> В книгах часто встречается множество кусочно-линейных аппроксимаций, неспособных ясно изобразить положение дел. К примеру, выражение:
>
> x = 1 для t < 0
> x = 0 для t >= 0
> было бы эквивалентно
> t 0< 1 AND
> как единому, а не кусочному выражению.

Я называю подобные флаги "гибридными величинами", поскольку они - булевские значения (значения "истина"), которые применяются в качестве данных (арифметических величин). Я тоже не знаю, как их еще назвать.

Можно сократить также и многочисленные ELSEы (когда оба результата - не нулевые), вычленяя различие между результатами.

К примеру,

```
: ШАГОВИКИ   'ПРОВЕРКА? @  IF  150  ELSE  151  THEN  LOAD ;
```

может быть упрощено до

```
: ШАГОВИКИ   150  'ПРОВЕРКА? @   1 AND +  LOAD ;
```

Этот подход работает потому, что, по смыслу, мы хотим загрузить либо блок 150, либо, для тестирования, следующий после него блок.

ЗАМЕЧАНИЕ О ТРЮКАХ.

Такого рода подходы часто клеймятся как "трюкачество". В большой компьютерной индустрии трюки имеют плохую репутацию.

Трюк - это просто использование преимуществ некоторых свойств операции. Трюки широко применяются в инженерной области. Дымоходы тянут дым, используя преимущества того факта, что тепло поднимается вверх. Автомобильные шины обеспечивают трение, используя преимущества земного притяжения. Арифметико-логические устройства (АЛУ) пользуются знанием того факта, что вычитание числа - это то же самое, что прибавление его двоичного дополнения.

Эти трюки позволяют создавать более простые и эффективные конструкции. Их использование оправдано тем, что предположения, на которых они строятся, безусловно останутся в силе.

Использование трюков становится опасным тогда, когда они зависят от чего-то, что вполне может измениться или не защищено упрятыванием информации.

Кроме того, трюки трудны для чтения, когда те предположения, на которых они основаны, не понятны или не объяснены. Что до замены условных операторов ANDами, то, когда эта технология становится частью словаря программиста, код может стать даже `более` читабельным. В случае применения трюка, специфичного для конкретного приложения, например, порядка, в котором данные должны размещаться в таблице, в листинге должны быть ясно задокументированы приемы, примененные в трюке.

СОВЕТ

Используйте MIN и MAX в качестве ограничителей.

Предположим, мы хотим уменьшить содержимое переменной ЧИСЛО, но не хотим, чтобы ее значение становилось меньше нуля:

```
-1 ЧИСЛО +!  ЧИСЛО @  -1 = IF  0 ЧИСЛО !  THEN
```

Это проще записать как

```
ЧИСЛО @  1-  0 MAX  ЧИСЛО !
```

В этом случае условный оператор заключен в слове MAX.

ИСПОЛЬЗОВАНИЕ ТАБЛИЦ РЕШЕНИЙ.

СОВЕТ
Используйте таблицы решений.

Мы предложили их во второй главе. Таблица решений – это структура, содержащая либо данные ("таблица данных"), либо адреса функций ("таблица функций"), сгруппированные в соответствии с любым числом измерений. Каждое из измерений представляет все возможные, взаимно исключающие состояния определенного аспекта проблемы. На пересечении "правильных" положений по каждому из измерений лежит нужный элемент: кусок данных или функция, которую надо выполнить.

Очевидно, что таблица решений для случая, когда задача живет в нескольких измерениях - это вариант лучший, нежели структура управления.

ТАБЛИЦА ДАННЫХ С ОДНИМ ИЗМЕРЕНИЕМ.

Вот пример простой одноразмерной таблицы данных. В нашей задаче имеется флаг по имени 'ШОССЕ?, который находится в "истине", когда мы имеем в виду загородные шоссе, и во "лжи", когда это - улицы города.

Давайте построим слово ОГРАНИЧЕНИЕ-СКОРОСТИ, возвращающее скоростной предел, зависящий от текущего состояния. Используя IF THEN, мы могли бы написать:

```
: ОГРАНИЧЕНИЕ-СКОРОСТИ  ( -- ограничение-скорости )
        'ШОССЕ? @ IF  55  ELSE  25  THEN ;
```

Мы можем устранить IF THEN, используя гибридное значение вместе с AND:

```
: ОГРАНИЧЕНИЕ-СКОРОСТИ   25  'ШОССЕ? @  30 AND + ;
```

Но такой метод не отвечает нашей концептуальной модели задачи и по этой причине не очень удобочитаем.

Давайте попробуем таблицу данных. Она имеет одно измерение и только два элемента, так что она не очень большая:

```
CREATE ПРЕДЕЛЫ  25 , 55 ,
```

Слово ОГРАНИЧЕНИЕ-СКОРОСТИ? должно теперь использовать булевское значение для подсчета смещения в таблице данных:

```
: ОГРАНИЧЕНИЕ-СКОРОСТИ  ( -- ограничение-скорости )
        ПРЕДЕЛЫ  'ШОССЕ? @  2 AND +  @ ;
```

Достигли ли мы чего-нибудь по сравнению с использованием IF THEN? Видимо, для такой простой задачи, нет.

Что мы все-таки сделали, так это отделили процесс принятия решения от самих данных. Это лучше окупается, если мы имеем более, чем один набор данных для одного решения. Предположим, мы имели бы еще

```
CREATE #ПОЛОС  4 , 10 ,
```

представляющее число полос на городской улице и на магистральном шоссе. Мы можем использовать идентичный код для вычисления текущего числа полос:

```
: #ПОЛОС?  ( -- #полос)
          #ПОЛОС  'ШОССЕ? @  2 AND  +  @ ;
```

Применяя технику факторизации, мы упрощаем это до:

```
: У-ДОРОГИ  ( для-шоссе для-города )  CREATE  , ,
        DOES>  ( -- данные)  'ШОССЕ? @  2 AND +  @ ;

        55 25 У-ДОРОГИ ОГРАНИЧЕНИЕ-СКОРОСТИ?
        10  4 У-ДОРОГИ #ПОЛОС?
```

ТАБЛИЦА ДАННЫХ С ДВУМЯ ИЗМЕРЕНИЯМИ.

Во второй главе мы представили задачу вычисления платы за телефон. На рис. 8-4 показано решение задачи с использованием двухмерной таблицы данных.

Рис.8-4. Решение задачи о плате за телефон.

```
\ Телефонные тарифы                                              03/30/84
CREATE ПОЛНЫЙ          30 ,     20 ,     12 ,
CREATE СРЕДНИЙ         22 ,     15 ,     10 ,
CREATE НИЗКИЙ          12 ,      9 ,      6 ,
VARIABLE ТАРИФ         \ показывает на ПОЛНЫЙ, СРЕДНИЙ или НИЗКИЙ
                       \ в зависимости от времени суток
ПОЛНЫЙ ТАРИФ ! \ к примеру
: ПЛАТА  ( о -- )  CREATE ,
        DOES>  ( -- плата )  @  ТАРИФ @ +  @ ;
0 ПЛАТА 1МИНУТА        \ плата за первую минуту
2 ПЛАТА +МИНУТА        \ плата за каждую дополнительную минуту
4 ПЛАТА /МИЛИ          \ плата за каждые 100 миль

\ Телефонные тарифы                                              03/30/84
VARIABLE ОПЕРАТОР?  \ 90, если помогал оператор, иначе 0
VARIABLE #МИЛЬ         \ сотни миль
: ?ПОМОЩЬ              ( прямой-вызов плата -- полная-плата)
        ОПЕРАТОР? @  + ;
: РАССТОЯНИЕ  ( -- плата )  #МИЛЬ @  /МИЛЬ * ;
: ПЕРВАЯ  ( -- плата )  1МИНУТА  ?ПОМОЩЬ  РАССТОЯНИЕ + ;
: ДОПОЛНИТЕЛЬНАЯ  ( -- плата )  +МИНУТА  РАССТОЯНИЕ + ;
: СУММА  ( #минут -- полная-плата)
```

1- ДОПОЛНИТЕЛЬНАЯ * ПЕРВАЯ + ;

В этой задаче каждое из измерений таблицы данных состоит из трех взаимно исключающих состояний. Поэтому простое булевское число (истина/ложь) здесь не подходит. Каждое измерение задачи реализовано особым образом.

Текущий тариф, который зависит от времени дня, хранится как адрес, представляющий одну из трех подтаблиц структуры тарифов. Мы можем сказать:

ПОЛНЫЙ ТАРИФ !

или

НИЗКИЙ ТАРИФ !

и т.д.

Текущая плата, будь то за первую минуту, за последующие или за расстояние, выражается как смещение в таблице (0, 2 или 4).

Замечание по оптимизации: мы разработали двухразмерную таблицу как набор из трех одноразмерных, указываемых ТАРИФом. Этот метод устраняет надобность в умножении, которое иначе бы потребовалось для построения структуры с двумя измерениями. В определенных случаях умножение может работать недопустимо медленно.

ТАБЛИЦА РЕШЕНИЙ С ДВУМЯ ИЗМЕРЕНИЯМИ.

Мы возвращаемся вновь к нашему примеру Крошечного Редактора из третьей главы для иллюстрации таблицы решений о двух измерениях.

На рисунке 8-5 мы конструируем таблицу функций, которые следует запускать, когда нажимаются различные клавиши. Результат то же, что и при применении оператора CASE, но только для двух имеющихся режимов - нормального и режима вставки. Каждая клавиша имеет различное поведение в зависимости от текущего режима.

Первый блок определяет смену режимов. Если мы вызываем

НОРМАЛЬНЫЙ РЕЖИМ# !

то переходим в нормальный режим.

ВСТАВОЧНЫЙ РЕЖИМ# !

вводит режим вставки.

Следующий блок строит таблицу функций по имени ФУНКЦИИ. Она состоит из кода ASCII клавиши, за которым следует адрес программы, вызываемой в нормальном режиме, а дальше – адрес программы, вызываемой в режиме вставки, когда на эту клавишу нажимают. После этого идет следующая клавиша со своей парой адресов, и так далее.

На третьем блоке слово 'ФУНКЦИИ берет значение клавиши, ищет его в таблице ФУНКЦИИ, а затем возвращает адрес ячейки, отвечающей найденному коду. (Мы предустанавливаем значение переменной СОВПАЛ на точку последней строки из таблицы - на те функции, которые надо выполнить, когда нажата `любая` клавиша.)

Слово ДЕЙСТВИЕ вызывает 'ФУКНЦИИ, затем добавляет содержимое переменной РЕЖИМ#. Поскольку эта переменная содержит либо 2, либо 4, добавляя

такое смещение, мы указываем внутри таблицы на программу, которую надо исполнить. Простое

```
@ EXECUTE
```

ее выполнит (или слово @EXECUTE или PERFORM, если оно есть в Вашей системе).

В Фиг-Форте надо изменить определение слова ЕСТЬ так:

```
: ЕСТЬ   [COMPILE] '  CFA , ;
```

Рис.8-5. Реализация Крошечного Редактора.

Блок # 30

```
\ Крошечный редактор
2 CONSTANT НОРМАЛЬНЫЙ  \ смещение в ФУНКЦИЯх
4 CONSTANT ВСТАВОЧНЫЙ  \        "
6 CONSTANT /КЛ                 \ байтов в таблице на каждую клавишу
VARIABLE РЕЖИМ#                \ текущее смещение в таблице
НОРМАЛЬНЫЙ РЕЖИМ# !
: ВЫКЛ-ВСТАВКУ          НОРМАЛЬНЫЙ РЕЖИМ# ! ;
: ВКЛ-ВСТАВКУ           ВСТАВОЧНЫЙ РЕЖИМ# ! ;

VARIABLE ВЫХОД?                \ t=время выходить из цикла
: ВЫХОД    TRUE ВЫХОД? ! ;
```

Блок # 31

```
\ Крошечный редактор          таблица функций
: ЕСТЬ  ' , ;           \   функция   ( -- )  ( для стандарта '83)
CREATE ФУНКЦИИ
\ клавиши                  нормальный           втставочный
 4 , ( Ctrl-D)             ЕСТЬ СТЕРЕТЬ         ЕСТЬ ВЫКЛ-ВСТАВКУ
 9 , ( Ctrl-I)        ЕСТЬ ВКЛ-ВСТАВКУ          ЕСТЬ ВЫКЛ-ВСТАВКУ
 8 , ( забой )             ЕСТЬ НАЗАД           ЕСТЬ ВСТАВИТЬ<
60 , ( стрелка влево)      ЕСТЬ НАЗАД           ЕСТЬ ВЫКЛ-ВСТАВКУ
62 , ( стрелка вправо)     ЕСТЬ ВПЕРЕД          ЕСТЬ ВЫКЛ-ВСТАВКУ
27 , ( выход)              ЕСТЬ ВЫХОД           ЕСТЬ ВЫКЛ-ВСТАВКУ
 0 , ( нет совпадения)     ЕСТЬ ЗАМЕСТИТЬ ЕСТЬ ВСТАВИТЬ
HERE /КЛ -  CONSTANT 'НЕ              \ адрес клавиши несовпадения
```

Блок # 32

```
\ Крошечный редактор  продолж.
VARIABLE СОВПАЛ
```

```
: 'ФУНКЦИИ  ( клавиша -- адр-совпадения )
  'НЕ СОВПАЛ !  'НЕ ФУНКЦИИ
  DO  DUP I @ =  IF  I СОВПАЛ !  LEAVE  THEN  /КЛ
  +LOOP  DROP  СОВПАЛ @ ;
: ДЕЙСТВИЕ  ( клавиша )  'ФУНКЦИИ  РЕЖИМ# @ +  @ EXECUTE ;
: ПУСК    FALSE ВЫХОД? !
   BEGIN  KEY ДЕЙСТВИЕ  ВЫХОД? @ UNTIL ;
```

В Фортах 79-го стандарта используйте:

```
: ЕСТЬ   [COMPILE] ' , ;
```

Мы устранили также избыточность при компиляции в определении сразу после таблицы функций:

```
HERE /КЛ -  CONSTANT 'НЕ  \ адрес клавиши несовпадения
```

Мы сделали константу для последней строки в таблице. (В тот момент, когда вызывается HERE, оно указывает на следующую свободную ячейку после последнего элемента таблицы. Последняя строка таблицы начинается на 6 байтов раньше.) Теперь имеются два слова:

ФУНКЦИИ (адрес начала таблицы функций)
'НЕ   (адрес строки "не-совпадения"; это программы, вызываемые для любой клавиши, которой нет в таблице)

Мы используем эти имена для получения адресов, передаваемых слову DO:

```
'НЕ ФУНКЦИИ DO
```

для запуска цикла, который пробегает от начала таблицы до ее конца. Нам неизвестно, сколько строк записано в этой таблице. Мы даже можем добавить или убрать строку в ней без необходимости изменения какого-либо другого места кода, даже того, где производится поиск в таблице.

Точно также константа /КЛ упрятывает информацию о количестве колонок в таблице.

К несчастью, подход, избранный для реализации слова 'ФУНКЦИИ упрощен и нехорош; в нем используется локальная переменная для уменьшения манипуляций со стеком. Более простое решение без использования локальной переменной:

```
: 'ФУНКЦИИ  ( клавиша -- адр-совпадения )
        'НЕ SWAP  'НЕ ФУНКЦИИ
        DO  DUP  I @ = IF  SWAP DROP  I SWAP  LEAVE  THEN
        /КЛ +LOOP  DROP ;
```

(Мы предложим другое решение попозже в этой главе, в "Использовании структурированных выходов".)

ТАБЛИЦЫ РЕШЕНИЙ ДЛЯ ПОВЫШЕНИЯ СКОРОСТИ.

Мы установили, что если можно вычислить значение вместо поиска его по таблице, то так и следует делать. Исключение составляет случай, когда требования по скорости оправдывают большую сложность таблицы. Вот пример вычисления степеней двойки с восьмиразрядной точностью:

```
CREATE ДВОЙКИ
        1 C,  2 C,  4 C,  8 C,  16 C,  32 C,
: 2**  ( n -- 2-в-степени-n )
        ДВОЙКИ +  C@ ;
```

Вместо вычисления ответа при помощи умножения двойки на саму себя "n" раз, все такие ответы вычислены заранее и записаны в таблицу. Можно просто добавлять смещение в таблице и получать результат.

В целом сложение работает гораздо быстрее умножения.

Мур приводит другой пример:

> Если Вам требуется подсчитывать кусочные функции, скажем, для графического дисплея, то обычно для этого высокого разрешения не нужно. Скорее всего, 7-ми битной функции будет совершенно достаточно. Табличный просмотр 128-ми чисел происходит быстрее, чем что бы то ни было другое из того, что Вы способны применить. Для вычисления низкочастотных функций таблицы решений превосходны.
>
> Однако если Вы вынуждены интерполировать, это значит, что приходится все равно вычислять функцию. Скорее всего, будет лучше подсчитать чуть более сложную функцию, избегая таким образом просмотра таблицы.

ПЕРЕПРОЕКТИРОВАНИЕ.

СОВЕТ

Одно изменение внизу способно предотвратить десять решений наверху.

В нашем интервью с Муром в начале главы мы отметили, что большое количество проверок могло бы быть устранено из задачи, если перепроектировать ее так, чтобы можно было применять два слова вместо одного: "Либо говоришь РАБОТАТЬ, либо ДЕЛАТЬ-ВИД".

Легче следовать простому, содержательному алгоритму при изменении контекста его окружения, чем выбирать из нескольких алгоритмов при одинаковом окружении.

Припомните наш пример слова ЯБЛОКИ из первой главы. Изначально оно было определено как переменная; затем на него по всей задаче была сделана масса ссылок из слов, которые увеличивали количество яблок (когда приходили новые партии), уменьшали его (когда яблоки продавались) и проверяли их текущее количество (при инвентаризации).

Когда стало необходимо поддерживать второй сорт яблок, `неправильный` подход состоял бы в том, чтобы добавить сложность во все слова по поставкам/продажам/проверкам. `Правильный` же подход был тот, который мы на самом деле выбрали: добавить сложность "внизу", то есть в сами ЯБЛОКИ.

Этот принцип может быть осуществлен многими способами. В главе 7 (в "Таблице состояний") мы употребляли таблицы состояний для реализации слов РАБОТАТЬ и ДЕЛАТЬ-ВИД, которые меняли значение группы переменных. Позже в этой главе мы использовали векторизованное исполнение для определений ВИДИМЫЙ и НЕВИДИМЫЙ с целью изменения значений слов TYPE', EMIT', SPACES' и CR' и легкого управления всем форматирующим кодом, который их использует.

СОВЕТ
Не делайте проверок для чего-то, что не может произойти.

Много современных программистов осчастливлены задачей проверки ошибок.

У функции нет необходимости проверять аргументы, передаваемые ей другим компонентом системы. Вызывающая программа должна сама нести ответственность за обеспечение нужных границ для вызываемого компонента.

СОВЕТ
Перепроверьте алгоритм.

Мур:

Большое количество условных операторов возникают из-за беспорядочного мышления при решении задачи. В теории сервоуправления многие думают, что алгоритм управления должен быть разным в зависимости от того, велико расстояние или мало. Издалека Вы работаете в чуть живом режиме, поближе - в замедленном, совсем близко - в торопливом. Приходится проверять свое удаление для выяснения того, какой из алгоритмов избрать.

Я выработал нелинейный алгоритм сервоуправления, который поддерживает весь этот диапазон. Этот подход устраняет перескоки в точках перехода из одного режима в другой. Он также убирает логику, нужную для принятия решения об алгоритме. Он освобождает Вас от необходимости эмпирически определять эти переходные точки. И, конечно, Ваша программа значительно упрощается, когда вместо трех алгоритмов применяется один. Вместо того, чтобы пытаться избавиться от условных операторов, лучше проверить теорию, которая приводит к возникновению этих операторов.

СОВЕТ
Остерегайтесь проверок на специальные случаи.

Один из примеров мы упомянули в этой книге раньше: если Вы обеспечиваете непопадание пользователя в беду, то Вам не требуется постоянно проверять, не попал ли он все-таки в эту беду.

Мур:

Другой хороший пример - это написание ассемблеров. Очень часто, даже если код операции может и не иметь ассоциированного с ним номера регистра, можно многое упростить, делая вид, что такой регистр есть - скажем, Регистр 0. Решение упрощается, когда производятся арифметические действия над ненужными битовыми полями. Просто запишите в них нули и продолжайте вычисления, которые Вы могли бы избежать, делая проверки на ноль и обходя их. Это - другая ипостась принципа "безразличности". Если Вам безразлично, то дайте безразличное число и используте его.

Каждый раз, когда у Вас возникает особая ситация, попытайтесь найти алгоритм, для которого эта ситация становится нормальной.

СОВЕТ
Используйте свойства компонента.

Хорошо спроектированный компонент - аппаратный или программный - позволит Вам разработать соответствующий лексикон в чистой и эффективной манере. Набор символьной графики старого принтера Epson MX-80 (хотя ныне и устаревший) хорошо иллюстрирует эту точку зрения. На рисунке 8-6 показаны графические символы, производимые кодами ASCII от 160 до 233.

Рис.8-6. Набор символьной графики Epson MX-80.

160	161	162	163	164	165	166	167
168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183
184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199
200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215
216	217	218	219	220	221	222	223

Каждый из графических символов является различной комбинацией шести квадратиков, либо заполненных, либо пустых. Предположим, в нашей задаче мы хотим использовать эти символы для создания рисунка. Мы знаем для каждого символа, что хотим иметь в каждой из шести позиций - и должны выдать подходящий ASCII-код на принтер.

Не нужно слишком долго приглядываться для понимания того, что здесь применена очень четкая последовательность. Пусть мы имеем шестибайтную таблицу, в которой каждый байт представляет пиксел в последовательности:

ПИКСЕЛЫ

0	1
2	3
4	5

и пусть каждый байт содержит шестнадцатеричное FF, если он "включен", ноль - если "выключен", тогда вот как мало надо кода, чтобы вычислить нужный символ:

```
CREATE ПИКСЕЛЫ  6 ALLOT
: ПИКСЕЛ  ( i -- a )  ПИКСЕЛЫ + ;
: СИМВОЛ  ( -- граф-символ )
        160  6 0 DO  I ПИКСЕЛ C@  I 2** AND  +  LOOP ;
```

(Мы ввели слово 2** несколько советов назад.)

Нет необходимости принимать какие-нибудь решения в определении слова СИМВОЛ. Этот последний просто вычисляется.

Замечание: для использования того же алгоритма при переводе шести смежных точек на большую ось мы можем просто переопределить слово ПИКСЕЛ. Это пример добавления перенаправления задним числом и хорошей декомпозиции.

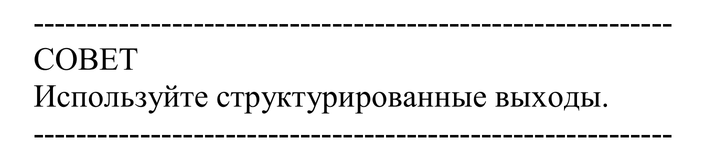
К сожалению, внешние компоненты не всегда хорошо спроектированы. К примеру, персональный компьютер IBM использует в своем дисплее аналогичную схему графических символов, но без какой-нибудь выраженной взаимосвязи между кодами ASCII и наборами точек. Единственный способ найти нужный символ - это сравнение наборов в просмотровой таблице.

Мур:

> Ассемблер для 68000 - это еще один пример, об который можно сломать голову в поисках того, как бы получше выразить эти коды операций минимальным набором команд. По всем признакам, хорошего решения нет. Люди, проектировавшие 68000, вообще не думали ни о каких ассемблерах. А ведь они могли бы гораздо облегчить положение вещей, причем задаром.

Если свойства компонента используются подобным образом, то код становится зависимым от этих свойств и от самого компонента. Впрочем, это простительно, поскольку весь зависимый код собран в единый лексикон, который может быть при необходимости легко изменен.

ИСПОЛЬЗОВАНИЕ СТРУКТУРИРОВАННЫХ ВЫХОДОВ.



СОВЕТ

Используйте структурированные выходы.

В главе по факторизации мы демонстрировали возможность вычленения структур управления при помощи следующего приема:

```
: УСЛОВНО    А Б OR  В AND  IF  NOT R> DROP  THEN ;
: АКТИВНЫЙ   УСЛОВНО  БЕСИТЬСЯ РЕЗВИТЬСЯ ПРЫГАТЬ ;
: ПАССИВНЫЙ  УСЛОВНО  СИДЕТЬ  ЕСТЬ   СПАТЬ ;
```

Форт позволяет нам менять поток управления, непосредственно манипулируя стеком возвратов. (Для устранения сомнений см. "Начала программирования...", гл. 9.) Неосмотрительное использование этого трюка может привести к неструктурированному коду с неприятными побочными эффектами. Однако дисциплинированное использование структурированных выходов может в действительности упростить код и таким образом улучшить его читабельность и управляемость.

Мур:

Все больше и больше мне нравится применять R> DROP для изменения потока управления. По эффекту это похоже на слово ABORT", котрое имеет встроенную конструкцию IF THEN. Но ведь там всего один IF THEN для всей системы, а не для каждой ошибки.

Я либо делаю выход, либо не делаю. Если я его делаю, то мне незачем проделывать свой остальной путь до конца. Я урезаю все целиком.

Альтернативой является обременение всей остальной задачи проверками на возникновение ошибки. Это неудобно.

"Аварийный выход" обманывает нормальный путь потока управления при определенных условиях. Форт дает эту возможность при помощи слов ABORT" и QUIT.

"Структурированный выход" расширяет эту концепцию, позволяя немедленное прекращение выполнения одного слова, без завершения задачи в целом.

Эта технология не должна путаться с использованием GOTO, которое неструктурировано донельзя. При помощи GOTO можно перейти куда угодно, внутрь или наружу текущего модуля. При нашем же методе Вы эффективно перепрыгиваете прямо на точку выхода из модуля (определения через двоеточие) и продолжаете исполнение вызывающего слова.

Слово EXIT заканчивает определение, в котором он появляется. Фраза R> DROP завершает определение, которое `вызвало` то определение, в котором эта фраза звучит; таким образом, оно производит тот же эффект, только может использоваться на один уровень ниже. Вот несколько примеров для обоих подходов.

Если у Вас имеется фраза с IF ELSE THEN, в которой после THEN больше ничего нет, типа:

```
... ГОЛОДЕН?  IF  C'ЕСТЬ  ELSE  ЗАКОПАТЬ  THEN ;
```

то можно удалить ELSE, используя EXIT:

```
... ГОЛОДЕН?  IF  C'ЕСТЬ EXIT  THEN  ЗАКОПАТЬ ;
```

(если условие истинно, мы едим и продолжаем; слово EXIT работает наподобие ;. Если условие ложно, мы перепрыгиваем на THEN и ЗАКОПАТЬ.)

Использование EXIT здесь более эффективно, оно экономит два байта и необходимость исполнения дополнительного кода, но оно же и ухудшает читабельность.

Мур комментирует ценности и опасности этой технологии:

> Когда работаешь с условными операторами, бывает особенно удобным выскакивать из середины без переходов на все эти THENы в конце. В одной задаче у меня было слово, выглядевшее так:
>
> ```
> : ПРОВЕРКА
>          ПРОСТОЙ   1УСЛОВИЕ IF  ... EXIT  THEN
>                    2УСЛОВИЕ IF  ... EXIT  THEN
>                    3УСЛОВИЕ IF  ... EXIT  THEN ;
> ```
>
> ПРОСТОЙ использовался для простых случаев, он заканчивался на R> DROP. Остальные условия были более сложными. Все выходят на ту же точку без необходимости болезненного прохождения через все IFы, ELSEы и THENы. В конечном результате, если ни одно из условий не выполнялось, стоял выход на ошибку.
>
> Это был плохой код, трудный в отладке. Однако он отобразил природу проблемы. Не было никакой лучшей схемы для его реализации. EXIT и R> DROP, по крайней мере, позволяли сохранить контроль над обстановкой.

Программисты иногда используют EXIT также для того, чтобы изящным образом выбираться из сложных циклов, использующих BEGIN. Или можно было бы использовать родственный метод для цикла DO LOOP, который мы использовали в слове 'ФУНКЦИИ в нашем Крошечном Редакторе раньше в этой главе. В этом слове мы просматриваем последовательность ячеек в поисках совпадения. Если мы таковое находим, то хотим вернуть адрес совпадения; если же не находим, то желаем иметь адрес последней строки в таблице функций.

Можно предложить слово LEAP (см. приложение В), которое будет работать как EXIT (будет имитировать ;). Теперь можно написать:

```
: 'ФУНКЦИИ  ( клавиша -- адр-совпадения )
        'НЕ ФУНКЦИИ
        DO  DUP  I @ = IF  DROP  I LEAP  THEN
        /КЛ +LOOP  DROP  'НЕ ;
```

Если совпадение находится, мы делаем LEAP, но не через +LOOP, а прямо из определения, оставляя I (адрес, по которому произошло совпадение) на стеке. Если оно не находится, мы проходим через ограничитель цикла и выполняем

DROP 'НЕ

что сбрасывает номер клавиши, которую искали, а затем оставляет адрес последней строки!

Как мы увидели, встречаются случаи, когда преждевременный выход хорошо подходит, и хороши даже многочисленные точки выхода и продолжения. Следует, однако, помнить, что использование EXIT и R> DROP в строгом смысле `не есть следование` принципам структурированного программирования и требует огромной осторожности.

К примеру, у Вас на стеке может быть число, которое в конце определения поглощается. Ранний EXIT оставит нежелательное число на стеке.

Забавы со стеком возвратов сродни играм с огнем. Можно обжечься. Но зато иметь огонь так здорово!

СКОРОСТНАЯ РАБОТА.

СОВЕТ
Выполняйте действие сразу, как только узнаете, что его
выполнить необходимо, но не позже.

Всякий раз, когда Вы устанавливаете флаг, спросите себя, зачем. Если ответ состоит в том, "чтобы знать, что мне надо сделать то-и-другое попозже", то задумайтесь, нельзя ли сделать то-и-другое `теперь`. Небольшое изменение структуры может существенно упростить Вашу разработку.

СОВЕТ
Не откладывайте на время исполнения того, что можете
сделать сегодня.

Когда у Вас есть возможность принять решение до компиляции, принимайте его.

Предположим, у Вас два варианта массивов: один с проверкой границ для стадии отладки, а другой, работающий быстрее, но без проверок, для реальной задачи.

Держите эти две версии на разных блоках. При компиляции задачи загружайте только нужную Вам версию.

Кстати, если Вы последуете этому предложению, то, видимо, замучаетесь убирать и добавлять скобочки в блоках загрузки для выбора всякий раз нужной версии. Вместо этого пишите определения, которые принимают решения за Вас. Пример (уже рассмотренный в другом контексте):

```
: ШАГОВИКИ   150  'ПРОВЕРКА? @  1 AND +  LOAD ;
```

СОВЕТ

Делайте с флагом действие "DUP", а не повторяйте его вычисление.

Иногда флаг нужен для того, чтобы показать, был или не был исполнен предыдущий участок кода. Вот определение, которое оставляет флаг, означающий, что СДЕЛАТЬ-ЭТО было исполнено:

```
: Я-СДЕЛАЛ?  ( t=сделал)
        Я-ДОЛЖЕН? IF  СДЕЛАТЬ-ЭТО  TRUE  ELSE  FALSE  THEN ;
```

Это можно упростить до:

```
: Я-СДЕЛАЛ?  ( t=сделал)
        Я-ДОЛЖЕН? DUP  IF  СДЕЛАТЬ-ЭТО  THEN ;
```

СОВЕТ
Не устанавливайте флаг, устанавливайте данные.

Если единственной целью установки флага является возможность для некоторого кода выбрать одно из двух чисел, то лучше устанавливать само такое число.

Пример "цветов" в шестой главе из раздела "Критерии для фрагментации" иллюстрирует этот тезис.

Задачей слова СВЕТЛО является установка флага, который показывает, не хотим ли мы установить бит повышенной яркости. Почему бы нам было не написать

```
: СВЕТЛО    TRUE 'СВЕТ? ! ;
```

для установки флага, и

```
'СВЕТ @ IF  8 OR  THEN ...
```

для его использования? Этот подход не столь прост, как элементарная запись в переменную битовой маски для интенсивности:

```
: СВЕТЛО   8 'СВЕТ? ! ;
```

и затем ее использование в виде

```
'СВЕТЛО @  OR ...
```

СОВЕТ
Не устанавливайте флаг, устанавливайте функцию.
(Векторизуйте.)

Этот совет аналогичен предыдущему и имеет тот же ареал обитания. Если единственной целью записи флага является возможность в будущем для кода принять решение между одной или другой функцией, то лучше записывать адрес самой функции.

К примеру, код для передачи символа на принтер отличен от того, который выкидывает его на видеодисплей. В плохой программе было бы написано:

```
VARIABLE УСТРОЙСТВО  ( 0=видео | 1=принтер )
: ВИДЕО   FALSE УСТРОЙСТВО ! ;
: ПРИНТЕР   TRUE УСТРОЙСТВО ! ;
: TYPE  ( a # -- )  УСТРОЙСТВО @ IF
        ( ... код для принтера ... )  ELSE
        ( ... код для дисплея .... )  THEN ;
```

Это плохо потому, что Вы принимаете рещение каждый раз, когда печатаете строку.

Предпочтительное решение должно использовать векторизованное исполнение, например:

```
DOER TYPE  ( a # -- )
: ВИДЕО   MAKE TYPE  ( ... код для дисплея ... ) ;
: ПРИНТЕР   MAKE TYPE  ( ... код для принтера ... ) ;
```

Это лучше, поскольку слову TYPE не надо каждый раз решать, какой из кодов использовать, оно это уже знает.

(В многозадачной системе задачи для принтера и для монитора будут иметь каждая свою собственную копию исполнительного вектора для TYPE, хранимую в пользовательской переменной.)

Вышеприведенный пример показывает также и ограничения для этого совета. В нашей второй версии у нас не осталось простого способа узнать, какое установлено текущее устройство – принтер или дисплей. Нам это может быть нужно, например, для того, чтобы решить, очищать ли экран или выдавать символ перевода формата. Мы используем знание о состоянии повторно, поэтому наше правило более неприменимо.

Флаг позволяет сделать простой реализацию дополнительных операций, зависящих от состояния. В случае слова TYPE, тем не менее, нам нужна скорость работы. Мы так часто печатаем строки, что не можем при этом зря тратить время. Лучшим решением здесь может быть и установка функции TYPE, и загрузка флага:

```
DOER TYPE  ( a # -- )
: ВИДЕО   0 УСТРОЙСТВО !  MAKE TYPE
        ( ... код для дисплея ... ) ;
: ПРИНТЕР   1 УСТРОЙСТВО !  MAKE TYPE
        ( ... код для принтера ... ) ;
```

TYPE уже знает, какой код исполнять, остальные же определения будут использовать флаг.

Другая возможность - это написание слова, которое выбирает значение вектора слова TYPE типа DOER (указатель на текущий код) и сравнивает его с адресом слова ПРИНТЕР. Если оно меньше этого адреса, то мы используем программу для ВИДЕО, иначе - обращаемся к ПРИНТЕРу.

Если изменение состояния приводит к изменению небольшого количества функций, то все равно можно использовать DOER/MAKE. Вот определения трех операторов работы с памятью, которые можно выключать одновременно.

```
DOER !'  ( векторное ! )
DOER CMOVE'  ( векторное CMOVE)
```

```
DOER FILL'  ( векторное FILL )
: ЗАПИСЬ     MAKE !'  ! ;AND
             MAKE CMOVE'  CMOVE ;AND
             MAKE FILL'  FILL ;AND
: -ЗАПИСЬ    MAKE !'  2DROP ;AND
             MAKE CMOVE'  2DROP DROP ;AND
             MAKE FILL'  2DROP DROP ;AND
```

Однако, если количество необходимых к векторизации функций велико, предпочтительнее таблица состояний.

Венчает это правило "запасной структурированный выход", слово типа DOER, которое векторизуется для осуществления структурированного выхода.

```
DOER КОЛЕБАТЬСЯ  ( запасной выход )
: РАЗРЫВ    КОЛЕБАТЬСЯ  РАЗВОДИТЬСЯ ;

      ( ... Гораздо позже в тексте: )
: СМЯГЧИТЬСЯ   MAKE КОЛЕБАТЬСЯ   ДАРИТЬ-ЦВЕТЫ  R> DROP ;
```

По умолчанию КОЛЕБАТЬСЯ ни к чему не приводит. Если мы вызываем РАЗРЫВ, то дело кончается судом. Однако если СМЯГЧИТЬСЯ перед РАЗРЫВом, то мы пошлем букет цветов, а затем перепрыгнем сразу к точке с запятой, минуя суд так, что наш партнер(ша) о его возможности и не догадается.

Такой подход особенно подходит там, где завершение должно производиться с помощью функции, определенной гораздо позже в программе (при декомпозиции по возрастанию сложности). Увеличение сложности более раннего кода ограничено единственно запасным выходом в нужной точке.

УПРОЩЕНИЕ.

Этот совет я оставил напоследок, поскольку он служит примером награды за стремление к простоте.

СОВЕТ
Пытайтесь не хранить какие бы то ни было флаги в памяти.

Флаг на стеке сильно отличается от такового в памяти. На стеке флаги могут быть легко получены (опросом аппаратуры, вычислением или чем угодно), а затем употреблены структурами управления. Короткая жизнь без особых затруднений.

Но только запишите флаг в память, и увидите, что случится. В добавление к наличию самого флага, у Вас теперь добавились сложности по его размещению. Ведь оно должно быть:

- создано
- инициализировано (даже до того, как что-то действительно изменится)
- сброшено (то есть передача его определенной команде оставит флаг в нужном текущем состоянии)

Поскольку в памяти все флаги - переменные, то они нереентерабельны.

Пример случая, в котором нам следовало бы переосмыслить необходимость флага - это тот пример, который мы уже несколько раз рассматривали. В "цветах" мы предположили, что наилучшим синтаксисом будет:

СВЕТЛО СИНИЙ

то есть прилагательное СВЕТЛО предшествует названию цвета. Отлично. Но помните код, реализующий эту версию? Сравните его с простотой такого подхода:

```
0 CONSTANT ЧЕРНЫЙ          1 CONSTANT СИНИЙ
2 CONSTANT ЗЕЛЕНЫЙ         3 CONSTANT ЖЕЛТЫЙ
4 CONSTANT КРАСНЫЙ         5 CONSTANT ФИОЛЕТОВЫЙ
6 CONSTANT КОРИЧНЕВЫЙ      7 CONSTANT СЕРЫЙ
: СВЕТЛЫЙ  ( цвет -- цвет' )  8 OR ;
```

В этой версии мы пересмотрели синтаксис и теперь говорим

СИНИЙ СВЕТЛЫЙ

Мы выставили цвет, а затем смодифицировали его.

Необходимость в переменной отпала, так же, как и в коде для ее разыменования и в еще большем объеме кода для ее сброса впоследствии. И полученный текст столь прост, что его невозможно не понять.

Когда я первый раз написал эти команды, то выбрал подход, свойственный для английского (или русского) языка. "СИНИЙ СВЕТЛЫЙ" звучит задом наперед, что не совсем приемлемо. Это было еще до начала моих бесед с Чаком Муром.

Философия Мура убедительна:

> Я бы провел различие между хорошей читабельностью на английском языке и просто хорошей читабельностью. В других языках, типа испанского, прилагательные следуют за существительными. Нам следовало бы быть независимыми от деталей того, на каком языке мы размышляем сами. Все зависит от Ваших личных наклонностей: к простоте ли или к эмуляции английского. Английский язык не столь превосходен, чтобы нам было необходимо следовать ему раболепно.

Если бы я продавал свои "цвета" в пакете для графических рисовальщиков, то побеспокоился бы насчет создания флага. Однако при написании этих слов исключительно для собственного употребления, и если бы мне пришлось делать это вновь, я предпочел бы Мурово влияние и использовал бы "СИНИЙ СВЕТЛЫЙ".

ИТОГИ

Использование логики и условных операторов в качестве существенного структурного элемента в программировании ведет к переусложненному, трудно

управляемому и неэффективному коду. В этой главе мы обсуждали несколько путей для минимизации, оптимизации или устранения ненужных структур условного управления.

В качестве последнего замечания: Фортовские игры с условными переходами и флагами не доступны в большинстве других современных языков. Реально же японцы базируют свой проект компьютера пятого поколения на языке под названием ПРОЛОГ - ПРОграммирование в ЛОГике - на котором пишут исключительно в логических терминах. Было бы интересно посмотреть на построение боевых рядов, если бы мы поставили ребром такой вопрос:

с IFами или без IFов?

В этой книге мы рассмотрели шесть первых шагов цикла разработки программного обеспечения, исследуя философские вопросы проектирования и практические соображения по реализации робастных, эффективных, читабельных программ.

Мы не обсуждали оптимизацию, выверение, отладку, документирование, управление проектом, Фортовы инструменты для разработки, определения на ассемблере, использование и ограничения рекурсии, разработку многопрограммных приложений и целевую компиляцию.

Но это уже совсем другая история.

ЛИТЕРАТУРА

1. Charles Eaker, "Just in Case," FORTH Dimensions II/3, p.37

ДЛЯ ДАЛЬНЕЙШИХ РАЗМЫШЛЕНИЙ

У Вас имеется слово CHOOSE, которое принимает аргумент "n" и возвращает случайное число в пределах от 0 до n-1. Результат всегда положительный или нулевой. Можно использовать CHOOSE для выдачи флага: фраза

```
2 CHOOSE
```

дает случайный флаг 0 или 1 (ложь или истина).

Напишите фразу для получения случайного числа между 0 и 19 (включительно) \`или\` между -20 и 0.

ЭПИЛОГ

ВОЗДЕЙСТВИЕ ФОРТА

НА МЫШЛЕНИЕ

Форт подобен Дао: это Путь, и осознается он, когда ему следуешь. Хрупкость его есть его сила; его простота есть его направление. (Майкл Хэм, победитель конкурса Mountain View Press на описание Форта в двадцать пять слов или меньше.)

Чтобы помочь в извлечении чего-нибудь из философии Форта, я произвел опрос нескольких пользователей Форта по вопросам:

"Как Форт повлиял на Ваше мышление? Не обнаружили ли Вы, что применяете 'Фортообразные' принципы в других областях?"

Вот некоторые из ответов:

Марк Бернстейн является президентом фирмы Eastgate Systems Inc. в Кембридже, штат Массачусетс, и имеет докторскую степень от химического факультета Гарвардского университета.

> Впервые я повстречался с Фортом, когда работал в области лазерной химии. Я пытался построить довольно сложный контроллер для нового лазерного спектрометра. Первоначальные планы предусматривали большой зеленый ящик, полный электроники - Интерфейс. Никто раньше конкретно этот вид прибора не делал - поэтому мы им и занимались – и список того, что мы хотим от компьютера, менялся каждые несколько недель.
>
> Через несколько месяцев у меня были сотни страниц программ на ассемблере, три большие платы, набитые микросхемами и 70-штырьковый разъем Системной Шины. День за днем все становилось все более шатким и трудно уловимым. Надписи на печатных платах испарились, разъемы законтачили, а запутанность ассемблерного кода возросла еще больше.
>
> Форт был очевидным решением программной проблемы, поскольку давал приличное окружение, в котором можно было строить и поддерживать сложный и быстро меняющийся код. Но сущность хорошего стиля программирования на Форте состоит в искусстве разбиения процедур на полезные, самостоятельные слова. Идея слов Форта нашла неожиданное применение для разработки лабораторного оборудования. Вместо построения большого, монолитного, всеобъемлющего Интерфейса, я обнаружил, что строю кучу простых маленьких коробочек, которые работали во многом как слова Форта: они имели фиксированный набор стандартных входов и выходов, они выполняли всего по одной функции, они проектировались для соединения друг с другом без особых усилий, и они были достаточно просты, чтобы можно было сказать, что делает такая коробочка, просто поглядев на ее этикетку.
>
> ... Идея "расширения человека", я думаю, является сегодня зародышевой идеей в концепции разработки программного обеспечения. Это касается не обязательно только для Форт-разработок; огромное удовольствие от UNIX, по крайней мере, в дни его юности, доставляло то, что его можно было прочитать (поскольку он был написан на Си), понять (поскольку он был мал) и изменить (поскольку он был прост).

Форт разделяет эти преимущества, хотя он был создан для другого сорта задач. Поскольку Форт мал, и поскольку он предоставляет своим пользователям контроль над их машинами, то он позволяет и по-человечески управлять приложениями. Тоскливо наблюдать за тем, как ученые сидят перед лабораторным компьютером, играя в "двадцать-два-вопроса" с пакетным программным обеспечением. При правильном использовании Форт позволяет ученому учить компьютер вместо того, чтобы разрешать компьютеру учить ученого. Как в хоккее, где игрок ощущает клюшку расширением своей руки, так и Форт расширяется человеком и помогает выработать ощущение, что достижения или просчеты компьютера являются также и твоими.

Реймонд Э. Десси является профессором химии Вирджинского политехнического института и университета в Блексбурге, штат Вирджиния.

Когда я пытался понять природу и структуру языка Си, я обнаружил, что применяю свои знания по организации и методам Форта. Это позволило мне понять закрученные или труднопроходимяе места в разделах описания Си.

Я обнаружил, что подход Форта является идеальной платформой, на которой можно строить смысловое и образовательное обрамление для концепций других языков и операционных систем.

Джерри Бутелль является владельцем фирмы Nautilus Systems в Санта-Круз, в Калифорнии, поставляющей фирменный кросс-компилятор Nautilus Cross-compiler.

Форт изменил много сторон моего мышления. После изучения Форта я программировал на других языках, включая ассемблер, Бейсик и Фортран. Я обнаружил, что использую тот же вид декомпозиции, который мы использовали в Форте, в смысле создания слов и группирования их вместе. К примеру, для обработки строк я определил бы аналоги для CMOVE, -TRAILING, FILL и т.д. Более существенно то, что Форт воскресил мою веру в простоту. Большинство людей выходят на задачи со сложными инструментами. Но простые инструменты доступнее и полезнее.

Я пытаюсь упростить все стороны своей жизни. Вот моя любимая цитата китайского философа Лао Цзы: "Чтобы достичь знания, добавляй что-то каждый день; чтобы достичь мудрости, каждый день отбрасывай что-то".

ПРИЛОЖЕНИЕ А

ОБЗОР ФОРТА

(ДЛЯ НОВИЧКОВ)

СЛОВАРЬ.

На Форте (FORTH) говорят словами (и числами), которые отделяются друг от друга пробелами:

ЛАДОНЬ ОТКРЫТЬ РУКА ОПУСТИТЬ ЛАДОНЬ ЗАКРЫТЬ РУКА ПОДНЯТЬ

Подобные команды могут быть набраны прямо с клавиатуры или вначале набраны редактором на устройствах массовой памяти (в файлах на диске), а потом загружены ("LOAD").

Все слова, уже имеющиеся в системе или определенные пользователем, существуют в "словаре", связном списке. "Определяющие слова" используются для добавления новых имен в словарь. Одним из них является слово : (произносится "двоеточие"), которое используется для определения нового слова в терминах ранее определенных слов. Вот как можно было бы определить новое слово по имени ПОДЫМАТЬ:

```
: ПОДЫМАТЬ    ЛАДОНЬ ОТКРЫТЬ  РУКА ОПУСТИТЬ
          ЛАДОНЬ ЗАКРЫТЬ  РУКА ПОДНЯТЬ ;
```

Слово ; заканчивает определение. Новое слово ПОДЫМАТЬ может теперь быть использовано вместо длинной последовательности слов, составляющих определение.

Слова Форта могут быть вложены друг в друга неограниченно. Написание задачи на Форте состоит из построения все более мощных определений, таких, как приведенное, в терминах тех, что были определены ранее.

Другим определяющим словом является CODE ("код"), которое используется вместо двоеточия для определения команды в терминах машинных инструкций используемого процессора. Слова, определенные с помощью CODE с точки зрения пользователя неотличимы от тех, что определены через двоеточие. Определения через CODE если и нужны, то только в самых критичных ко времени исполнения местах программы.

СТРУКТУРЫ ДАННЫХ.

Еще одно определяющее слово - CONSTANT (конатанта) - используется так:

```
17 CONSTANT СЕМНАДЦАТЬ
```

Новое слово СЕМНАДЦАТЬ может теперь быть использовано вместо настоящего числа 17.

Определяющее слово VARIABLE (переменная) создает место для хранения временных данных. VARIABLE использется так:

VARIABLE БАНАНЫ

Создается место (ячейка) в памяти, идентифицируемое именем БАНАНЫ.

Получение содержимого этой области памяти - это задача слова @ (произносится "разыменовать" или "взять"). К примеру,

БАНАНЫ @

достает содержимое переменной БАНАНЫ. Его антонимом является слово ! (произносится "загрузить" или "записать"), которое загружает число в ячейку памяти, типа

100 БАНАНЫ !

В Форте есть также слово для увеличения текущего содержимого на заданное число; например, фраза

2 БАНАНЫ +!

увеличивает счетчик на два, делая его равным 102.

В Форте есть и много других операторов для структур данных но, что более важно, в нем содержатся также инструменты, нужные программисту для создания структур данных любого типа, требуемого в задаче.

СТЕК.

В Форте переменные и массивы служат для сохранения значений, которые могут быть нужны для множества программ и/или в непредсказуемые моменты времени. Они `не` используются для локальной передачи данных между определениями. Для этого Форт применяет гораздо более простой механизм: стек данных.

Когда Вы набираете число, оно кладется на стек. Когда Вы вызываете слово, имеющее числовой аргумент, то оно забирает его со стека. Так, фраза

17 SPACES

выдаст семнадцать пробелов на текущее устройство вывода. "17" кладет на стек двоичный эквивалент числа 17; слово SPACES его употребляет.

Константа также кладет на стек свое значение; так, фраза

СЕМНАДЦАТЬ SPACES

дает тот же эффект.

Стек работает на основе принципа "последним вошел – первым вышел" (LIFO). Это значит, что данные могут передаваться между словами упорядоченным, модульным образом, соответствующим вложенности определений через двоеточие.

К примеру, определение по имени ОСЬ могло бы вызывать фразу 17 SPACES. Это временное использование стека будет незаметно для любого другого определения, вызывающего ОСЬ, поскольку число, положенное на стек, снимается с него до того, как заканчивается определение слова ОСЬ. Такое вызывающее слово могло само положить некоторые свои числа на стек до вызова слова ОСЬ. Эти числа останутся на стеке без повреждений после того, как ОСЬ будет отработана и вызывающее определение продолжит свою работу.

СТРУКТУРЫ УПРАВЛЕНИЯ.

Форт предлагает все структуры управления, необходимые для структурированного программирования без использования GOTO.

Синтаксис конструкции IF THEN таков:

```
... ( флаг ) IF СТУЧАТЬ THEN ОТКРЫТЬ ...
```

"Флаг" - это число на стеке, которое употребляется частью IF. Ненулевое значение этого флага означает истину, а нулевое - ложь. Истинный флаг вызывает исполнение кода между IF (в данном случае, слова СТУЧАТЬ). Слово THEN отмечает конец фразы для условного исполнения; работа продолжается со слова ОТКРЫТЬ. Флаг со значением "ложь" дает `запрет` исполнения фразы между IF и THEN. В любом случае будет исполнено слово ОТКРЫТЬ.

Слово ELSE позволяет создавать альтернативные фразы для условного исполнения при ложном флаге. Во фразе

```
( флаг ) IF СТУЧАТЬ ELSE ЗВОНИТЬ THEN ОТКРЫТЬ ...
```

слово СТУЧАТЬ будет исполнено, если флаг истинен, в противном случае будет исполнено слово ЗВОНИТЬ. В любом из случаев работа будет продолжена, начиная со слова ОТКРЫТЬ.

Форт позволяет также создавать циклы со счетчиком в виде

```
( верх ) ( низ ) DO ... LOOP
```

или неопределенные циклы в формах

```
... BEGIN ... ( флаг ) UNTIL
```

и

```
... BEGIN ... ( флаг ) WHILE ... REPEAT
```

ГДЕ НАЙТИ ПОЛНОЕ ОПИСАНИЕ.

Полноценное введение в набор команд Форта можно прочитать в книге `Starting FORTH`, выпущенное издательством Prentice-Hall. (Эта книга выпущена на русском языке под названием "Начальный курс программирования на языке Форт" - М:Финансы и статистика, 1990.)

ПРИЛОЖЕНИЕ Б

ОПРЕДЕЛЕНИЕ DOER / MAKE

Если в Вашей системе слова DOER и MAKE еще не определены, это приложение призвано помочь Вам их ввести и, при необходимости, понять принцип их работы. Поскольку по природе своей эти конструкции системно-зависимы, я привел несколько различных реализаций в конце приложения в надежде, что одна из них будет работать и у Вас. Если же этого не произойдет, и если в этом разделе Вам не хватит информации для того, чтобы заставить все работать, то, видимо, у Вас какая-то необычная система. Пожалуйста, не обращайтесь за помощью ко мне; спросите поставщиков Вашего Форта.

Вот как это работает. DOER - это определяющее слово, которое создает словарную статью с одной ячейкой памяти в поле ее параметров. Эта ячейка содержит адрес вектора и инициализируется указанием на слово, которое ничего не делает (по имени NOTHING).

Потомки слова DOER исполняют код после DOES> в нем, который делает всего две вещи: достает адрес вектора и заносит его на стек возвратов. Это все. Продолжение исполнения Форта производится с этого адреса со стека возвратов, что вызывает исполнение векторизованной функции. Это все равно, что сказать (в стандарте '83)

' NOTHING >BODY >R <возврат-каретки>

что даст исполнение NOTHING. (Такой трюк годится только для определений через двоеточие.)

Вот иллюстрация по словарной статье, созданной после ввода

DOER ДЖО

ДЖО	pfa слова NOTHING
заголовок	поле параметров

Теперь предположим, мы определили

: TECT   MAKE ДЖО CR ;

то есть мы создали слово, которое может направить ДЖО на выдачу перевода каретки.

Вот рисунок, изображающий скомпилированное определение слова TECT:

TECT	адр. (MAKE)	0	адр. ДЖО	адр. CR	адр. EXIT
заголовок		маркер			

Давайте глянем на код для MAKE. Поскольку мы используем его внутри определения через двоеточие, переменная STATE будет в состоянии "истина", и мы исполним фразу

COMPILE (MAKE) HERE MARKER ! 0 ,

Можно видеть, как MAKE скомпилировало адрес программы времени исполнения (MAKE), после которого записала ноль. (Мы объясним, для чего этот ноль и почему мы записали его адрес в переменную MARKER, попозже.)

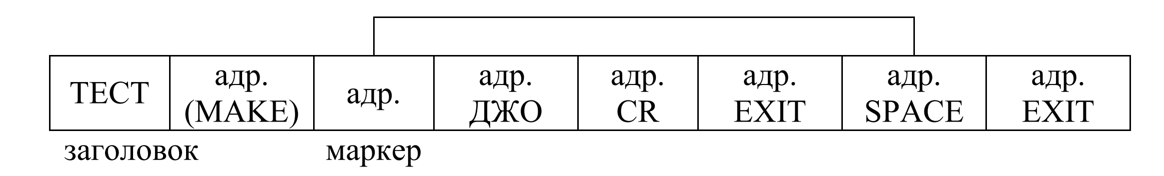
Теперь посмотрим, что (MAKE) делает, когда мы исполняем новое определение ТЕСТ:

R>	Получает адрес со стека возвратов. Этот адрес указывает на ячейку сразу после (MAKE), где находится ноль.
DUP 2+	Получает адрес следующей ячейки после (MAKE), где размещен адрес ДЖО.
DUP 2+	Получает адрес третьей ячейки после (MAKE), где начинается код, который мы хотим исполнить. На стеке теперь ('маркера 'джо 'кода)
SWAP @ >BODY	Берет содержимое адреса, указывающего на ДЖО (т.е. получает адрес самого ДЖО) и вычисляет pfa ДЖО, где хранится адрес вектора.
!	Записывает адрес, по которому начинается новый код (CR и т.д.) по адресу вектора ДЖО. Теперь ДЖО указывает внутрь определения слова ТЕСТ. Если мы введем ДЖО, мы получим возврат каретки.
@ ?DUP	Берет содержимое ячейки, содержащей ноль.
IF >R THEN	Поскольку там ноль, тело IF THEN не исполняется.

Вот основная идея. Но как насчет ячейки с нулем? Она – для использования слова ;AND. Предположим, мы изменили ТЕСТ так:

: ТЕСТ   MAKE ДЖО CR ;AND SPACE ;

То есть когда мы вызываем ТЕСТ, оно направит вектор ДЖО на CR, а затем немедленно исполнит SPACE. Вот как будет выглядеть новая версия ТЕСТ:



Вот определение ;AND:

: ;AND   COMPILE EXIT HERE MARKER @ ! ; IMMEDIATE

Видно, что ;AND скомпилировало EXIT так же, как это сделало бы слово ;.

Далее, припомните что MAKE сохранило адрес нуля в переменной MARKER. Теперь ;AND записывает HERE (место начала следующего участка кода, начинающегося со SPACE) в ячейку, которая содержала ноль. Теперь (MAKE) имеет указатель на место продолжения исполнения. Фраза

IF >R THEN

теперь положит на стек возвратов адрес кода, начинающегося со слова SPACE. Так выполнение перескочит через код между MAKE и ;AND и продолжится для остальной части определения через двоеточие.

Слово UNDO получает адрес слова-DOERа и записывает в него ссылку на слово NOTHING.

Одно последнее замечание: на некоторых системах может возникнуть проблема. Если Вы используете MAKE вне определения через двоеточие для создания ссылки вперед, то можете оказаться не в состоянии найти самое последнее из определенных слов. К примеру, если у Вас имеется

```
: ПРИПЕВ    ТРАМ- ПАМ- ПАМ- ;
MAKE ПЕСНЯ   КУПЛЕТ ПРИПЕВ ;
```

то Ваша система может подумать, что ПРИПЕВ еще не определен. Проблема заключается в месторасположении слова SMUDGE. В качестве решения попытайтесь перегруппировать порядок определений или, при необходимости, уберите код с MAKE внутрь определения, которое потом можно исполнить:

```
: УСТАНОВКА    MAKE ПЕСНЯ   КУПЛЕТ ПРИПЕВ ;  УСТАНОВКА
```

В системе Laboratory Microsystems PC/FORTH 2.0 слово UNSMUDGE в 9-й строке устраняет эту проблему. В модели Форта Лексена/Перри/Харриса этой проблемы нет.

Последний блок - это пример использования DOER/MAKE. После загрузки блока введите

```
RECITAL
```

а затем введите

```
WHY?
```

и возврат каретки столько раз, сколько захочется. (Всякий раз у Вас будет для этого своя причина.)

```
Блок # 21
0 ( DOER/MAKE          Теневой блок                        LPB 12/05/83 )
1 NOTHING              нет операции
2 DOER                 определяет слово с векторизуемым поведением
3 MARKER               хранит адрес служебного указателя продолжения
4 (MAKE)               устанавлиает адрес последующего кода в поле
5                      параметров слова типа DOER
6 MAKE                 интерпретация: MAKE doer-имя  Форт-код ;
7                      или внутри определения:
8                                : ОПР   MAKE doer-имя  Форт-код ;
9                      векторизует слово doer-имя на Форт-код.
10 ;AND                разрешает продолжение определения с MAKE.
11 UNDO                использование: UNDO doer-имя  делает его
12                     безопасным в использовании.
13
14
15
```

Блок # 22

```
\ DOER/MAKE  FORTH-83 Laxen/Perry/Harris                    LPB 12/05/83
: NOTHING ;
: DOER   CREATE  ['] NOTHING  >BODY ,  DOES> @ >R ;
VARIABLE MARKER
: (MAKE)  R>  DUP 2+  DUP 2+  SWAP @  >BODY !
      @ ?DUP IF >R THEN ;
: MAKE   STATE @ IF ( компиляция)
      COMPILE (MAKE)  HERE MARKER !  0 ,
      ELSE  HERE  [COMPILE] '  >BODY !
      [COMPILE] ]  THEN ; IMMEDIATE
: ;AND   COMPILE EXIT  HERE MARKER @ ! ; IMMEDIATE
: UNDO   ['] NOTHING  >BODY  [COMPILE] '  >BODY ! ;
\ Код в этом блоке является общественным достоянием.
```

Блок # 23

```
( DOER/MAKE  FORTH-83 LabMicro PC/FORTH 2.0  LPB 12/05/83 )
: NOTHING ;
: DOER   CREATE  ['] NOTHING  >BODY ,  DOES> @ >R ;
VARIABLE MARKER
: (MAKE)  R>  DUP 2+  DUP 2+  SWAP @  >BODY !
      @ ?DUP IF >R THEN ;
: MAKE   STATE @ IF ( компиляция)
      COMPILE (MAKE)  HERE MARKER !  0 ,
      ELSE  HERE  [COMPILE] '  >BODY !
      [COMPILE] ] UNSMUDGE  THEN ; IMMEDIATE
: ;AND   COMPILE EXIT  HERE MARKER @ ! ; IMMEDIATE
: UNDO   ['] NOTHING  >BODY  [COMPILE] '  >BODY ! ;
( Код в этом блоке является общественным достоянием.)
```

Блок # 24

```
( DOER/MAKE  FIG model                    LPB 10/25/84 )
: NOTHING ;
: DOES-APF  ( apf -- arf-потомка-<BUILDS-DOES> )  2+ ;
: DOER   <BUILDS  ' NOTHING ,  DOES> @ >R ;
VARIABLE MARKER
: (MAKE)  R>  DUP 2+  DUP 2+  SWAP @  2+ DOES-APF !
      @ -DUP IF >R THEN ;
: MAKE   STATE @ IF ( компиляция)
      COMPILE (MAKE)  HERE MARKER !  0 ,
      ELSE  HERE  [COMPILE] '  DOES-APF !
      SMUDGE   [COMPILE] ]  THEN ; IMMEDIATE
: ;AND   COMPILE ;S  HERE MARKER @ ! ; IMMEDIATE
: UNDO   ' NOTHING   [COMPILE] '  DOES-APF ! ;
;S
( Код в этом блоке является общественным достоянием.)
```

Блок # 25

```
( DOER/MAKE  Стандарт-79  MVP FORTH              LPB 12/05/83 )
: NOTHING ;
: DOER   CREATE  ' NOTHING ,  DOES> @ >R ;
VARIABLE MARKER
: (MAKE)  R>  DUP 2+  DUP 2+  SWAP @  2+ ( apf) !
        @ ?DUP IF >R THEN ;
: MAKE   STATE @ IF ( компиляция)
        COMPILE (MAKE)  HERE MARKER !  0 ,
        ELSE  HERE  [COMPILE] ' !
        [COMPILE] ]  THEN ; IMMEDIATE
: ;AND   COMPILE EXIT  HERE MARKER @ ! ; IMMEDIATE
: UNDO   ' NOTHING   [COMPILE] ' ! ;


( Код в этом блоке является общественным достоянием.)
```

Блок # 26

```
( Пример на DOER/MAKE                          12/27/84 )
DOER ANSWER
: RECITAL   CR
        ." Ваш папа стоит на столе. Спросите его 'WHY?' (почему)"
        MAKE ANSWER ." Для замены лампочки."
        BEGIN
        MAKE ANSWER ." Потому что она сгорела."
        MAKE ANSWER ." Потому что была старая."
        MAKE ANSWER ." Потому что мы ее привинтили очень давно."
        MAKE ANSWER ." Потому что было темно!"
        MAKE ANSWER ." Потому что стояла ночь!!"
        MAKE ANSWER ." Перестань спрашивать ПОЧЕМУ?"
        MAKE ANSWER ." Потому что я с тобой свихнусь."
        MAKE ANSWER ." Дай мне просто поменять эту лампочку!"
        FALSE UNTIL ;
: WHY?  CR ANSWER  QUIT ;
```

ПРИЛОЖЕНИЕ B

ДРУГИЕ УТИЛИТЫ ,

ОПИСАННЫЕ В ЭТОЙ КНИГЕ

Это приложение призвано помочь Вам определить некоторые из слов, которые упоминались в этой книге и которые могут отсутствовать в Вашей системе. Определения даны в Стандарте-83.

ИЗ ГЛАВЫ 4.

Определение слова ASCII для работы в Стандарте-83:

```
: ASCII  ( -- c) \ компиляция:   c   ( -- )
\ интерпретация:   c   ( -- c)
        BL WORD 1+  C@  STATE @
        IF  [COMPILE] LITERAL  TNEN ; IMMEDIATE
```

ИЗ ГЛАВЫ 5.

Слово \ можно определить как:

```
: \  ( пропустить остаток строки )
        >IN @  64 /  1+  64 *  >IN ! ; IMMEDIATE
```

Если Вы решили не использовать слово EXIT для прерывания интерпретации блока, то можете определить слово \S как

```
: \S   1024 >IN ! ;
```

Слово FH определяется просто как

```
: FH  \  ( смещение -- блок-по-смещению )
\ "from here" - "отсюда"
        BLK @ + ;
```

Подобная факторизация позволяет использовать FH многими способами, типа:

```
: TEST   [ 1 FH ] LITERAL  LOAD ;
```

или

```
: SEE   [ 2 FH ] LITERAL  LIST ;
```

Несколько более сложная версия этого слова позволяет также редактировать или загружать блоки фразами вида "14 FH LIST", работающими относительно последнего блока, который был напечатан LISTом (SCR):

```
: FH  \  ( смещение -- блок-по-смещению ) "from here" - "отсюда"
         BLK @ ?DUP 0= IF  SCR @  THEN  + ;
```

Слово BL - это просто константа:

```
32 CONSTANT BL
```

TRUE и FALSE могут быть определены так:

```
0 CONSTANT FALSE
-1 CONSTANT TRUE
```

(Слова структур управления Форта, такие, как IF и UNTIL, рассматривают нуль как "ложь" и любое ненулевое значение как "истину". До Форта-83 соглашение предусматривало показывать "истину" значением 1. Начиная с '83-го стандарта, однако, "истина" имеет значение FFFF (шестнадцатеричное), что соответствует числу со знаком -1 (все биты установлены).

Слово WITHIN на высоком уровне можно определить так:

```
: WITHIN  ( n низ верх+1 -- ? )
         OVER - >R  - >R  U< ;
```

ИЗ ГЛАВЫ 8.

Реализация слова LEAP будет зависеть от того, как Ваша система реализует циклы DO LOOP. Если DO держит два значения на стеке возвратов (индекс и ограничитель), LEAP должно их оба сбрасывать, плюс сбрасывать еще одно значение со стека возвратов для выхода:

```
: LEAP   R> R> 2DROP  R> DROP ;
```

Если DO держит `три` значения на стеке возвратов, то следует определить:

```
: LEAP   R> R> 2DROP  R> R> 2DROP ;
```

ПРИЛОЖЕНИЕ Г

ОТВЕТЫ НА ЗАДАЧИ

" ДЛЯ ДАЛЬНЕЙШЕГО РАЗМЫШЛЕНИЯ "

ГЛАВА 3.

1. Ответ зависит от того, считаете ли Вы, что другим компонентам надо будет "знать" числовое значение, связанное с каждой клавишей. Чаще этого `не` требуется. Простая, более компактная форма здесь поэтому предпочтительнее. Также в первой версии добавление нового кода клавиши потребует изменений в двух местах.
2. Проблема со словами RAM-ALLOT и THERE состоит в том, что они `зависимы по времени`: мы их должны исполнять в определенном порядке. Здесь нашим решением могло бы быть отделение от интерфейса указателя места RAM, который от порядка не зависит; это можно было бы сделать, имея `единственное` слово, которое прозрачно исполняло бы обе функции.
Синтаксис наших слов получился бы таким:

```
: RAM-ALLOT   ( #байтов-для-размещения -- начальный-адрес )
... ;
```

Этот синтаксис останется неизменным, если мы его поменяем для размещения снизу вверх:

```
: RAM-ALLOT   ( #байтов-для-размещения -- начальный-адрес )
        >RAM @  SWAP -  DUP >RAM ! ;
```

ГЛАВА 4.

Наше решение таково:

```
\ КАРТЫ              Перетасовка                           12-01-84
52 CONSTANT          #КАРТ
CREATE КОЛОДА  #КАРТ ALLOT                \ одна карта на байт
: КАРТА  ( i -- адр )  КОЛОДА + ;
: НОВАЯ-КОЛОДА
        #КАРТ 0 DO  I  I КАРТА C!  LOOP ;
НОВАЯ-КОЛОДА
: 'CSWAP  ( a1 a2 -- ) \ поменять байты по a1 и a2
        2DUP C@  SWAP C@  ROT C!  SWAP C! ;
: ТАСОВАТЬ  \  тасовать колоду карт
        #КАРТ 0 DO  I КАРТА  #КАРТ CHOOSE КАРТА  'CSWAP  LOOP ;
```

ГЛАВА 8.

Будет работать и это:

20 CHOOSE 2 CHOOSE IF NEGATE THEN

Но проще так:

40 CHOOSE 20 -

ПРИЛОЖЕНИЕ Д

СВОД СТИЛИСТИЧЕСКИХ СОГЛАШЕНИЙ

ОТСТУПЫ И ПРОПУСКИ.

1 пробел между : и именем
2 пробела между именем и его комментарием [1]
2 пробела или новая строка после комментария до тела определения
3 пробела между именем и телом определения, если комментарии не используются
3 пробела отступа для каждой из последовательных строк (или кратные тройке отступы для выделения вложенности)
1 пробел между словами/числами внутри фразы
2 или 3 пробела между фразами
1 пробел между последним словом и ;
1 пробел между ; и IMMEDIATE (при необходимости)

Не ставить пустых строк между определениями, кроме случаев разграничения существенных групп определений.

АББРЕВИАТУРЫ ДЛЯ СТЕКОВЫХ КОММЕНТАРИЕВ.

n	чило одинарной длины со знаком
d	число двойной длины со знаком
u	чило одинарной длины без знака
ud	число двойной длины без знака
t	тройная длина
q	учетверенная длина
c	7 (или 8)-битный символ
b	8-ми битный байт
?	булевский флаг, или:
t=	(true) истина
f=	(false) ложь
a или adr или адр	адрес

[1] Часто наблюдаемая альтернатива - 1 пробел между именем и комментарием и 3 - между комментарием и определением. Более либеральный подход использует по 3 пробела до и после комментария. Что бы Вы ни выбрали, будьте последовательны.

acf	адрес поля кода
apf	адрес поля параметров (в качестве префикса) адрес чего-либо
s d	(как пара) источник приемник
lo hi	нижняя- верхняя-граница (включительно)
#	число (количество)
o	(offset) смещение
i	индекс
m	маска
x	безразлично (для структур данных)

"Смещение" - это разница, выраженная в абсолютных единицах, например, байтах.

"Индекс" - это разница, выраженная в логических единицах, например, элементах записи.

КОММЕНТАРИИ ДЛЯ ВХОДНОГО ПОТОКА.

c	одиночный символ, выделенный пробелами
name или имя	последовательность символов, выделенная пробелами
text или текст	последовательность символов, выделенная не пробелами

После слова "текст" ставьте требуемый символ-ограничитель, типа: текст" или текст).

ПРИМЕРЫ ХОРОШЕГО СТИЛЯ КОММЕНТИРОВАНИЯ.

Вот два примерных блока для иллюстрации хорошего стиля написания примечаний.

Блок # 126

```
\ Форматер          Структуры данных -- стр.2               06/06/83
 6 CONSTANT TMARGIN          \ #строки начала тела текста
55 CONSTANT BMARGIN          \ #строки конца тела текста

CREATE HEADER  82 ALLOT
\ { 1счет-слева | 1счет-справа | 80заголовок }
CREATE FOOTER  82 ALLOT
\ { 1счет-слева | 1счет-справа | 80подпись }

VARIABLE ACROSS              \ текущ. горизонтальная поз. форматтера
VARIABLE DOWNWARD            \ текущ. вертикальная поз. форматтера
VARIABLE LEFT                \ текущ. начальная левая граница
VARIABLE WALL                \ текущ. начальная правая граница
VARIABLE WALL-WAS \ WALL при нач. форматирования тек. стр.
```

Блок # 127

```
\ Форматер           позиционирование -- стр.1                    06/06/83
: SKIP  ( n)  ACROSS + ;
: NEWLEFT           \ сбросить левую границу
  LEFT @  PERMANENT @ +  TEMPORARY @ +  ACROSS ! ;
: \LINE               \ начать новую строку
  DOOR  CR'  1 DOWNWARD +!  NEWLEFT  WALL @  WALL-WAS ! ;
: AT-TOP?  ( -- t=наверху)  TMARGIN  DOWNWARD @ = ;
: >TMARGIN  \ переместиться от crease до TMARGIN
  0 DOWNWARD !  BEGIN  \LINE  AT-TOP? UNTIL ;
```

СОГЛАШЕНИЯ ПО ФОРМИРОВАНИЮ ИМЕН.

Значение	Форма	Пример
Арифметика		
целое 1	1имя	1+
целое 2	2имя	2*
берет родственные входные параметры	+имя	+DRAW
берет масштабирующие входные параметры	*имя	*DRAW
Компиляция		
начало "высокоуровневого" кода	имя:	CASE:
конец "высокоуровневого" кода	;имя	;CODE
добавить что-либо в словарь	имя,	C,
исполняется при компиляции	[имя]	[COMPILE]
несколько отлично	имя' (штрих)	CR'
внутреннее представление или примитив	(имя) или <имя>	(TYPE) <TYPE>
часть периода исполнения компилирующего слова:		
где есть строчные буквы	строчными	if
где нет строчных букв	(ИМЯ)	(IF)
определяющее слово	:имя	:COLOR
номер блока с оверлеем	имяING	DISKING
Структуры данных		
таблица или массив	имена	ЗАНЯТЫЕ
общее число элементов	#имя	#ЗАНЯТЫХ
текущий номер (переменная)	имя#	ЗАНЯТЫЙ#
установить текущий номер (n) имя	13 ЗАНЯТЫЙ	
переход на следующий элемент	+имя	+ЗАНЯТЫЙ
размер смещения до записи от		

начала структуры	имя+	ДАТА+
размер (в байтах) (сокращение от БАЙТОВ/имя)	/имя	/ЗАНЯТОГО
указатель счетчика (индекс)	>имя	>IN
перевести адрес структуры в адрес поля (записи)	>имя	>BODY
индекс в файле	(имя)	(ЛЮДИ)
указатель в файле	-имя	-РАБОТА
инициализировать структуру	0имя	0ЗАПИСЬ

Направление, Преобразование

назад	имя<	СМЕСТИТЬ<
вперед	имя>	CMOVE>
от	<имя	<TAPE
до	>имя	>TAPE
преобразовать в	имя>имя	ФУТЫ>МЕТРЫ
вниз	\имя	\LINE
вверх	/имя	/LINE
открыть	{имя	{FILE
закрыть	}имя	}FILE

Логика, Управление

вернуть булевский флаг	имя?	КОРОТКИЙ?
вернуть обратный флаг	-имя?	-КОРОТКИЙ?
адрес булевского значения	'имя?	'КОРОТКИЙ?
работает условно	?имя	?DUP (быть может, DUP)
включить	+имя	+ЧАСЫ
или отсутствие символа	имя	МИГАНИЕ
выключить	-имя	-МИГАНИЕ

Память

сохранить значение	@имя	@КУРСОР
восстановить значение	!имя	!КУРСОР
записать в	имя!	СЕКУНДЫ!
считать из	имя@	СЧЕТЧИК@
имя буфера	|имя	|ВСТАВКИ
адрес имени	'имя	'S
адрес указателя на имя	'имя	'TYPE
обменять, особенно байты	>имя<	>MOVE<

Числовые типы

длиной в байт	Cимя	C@
длиной 2 ячейки в двоичном дополнительном целом коде	Dимя	D+
смешанное 16 и 32-х разрядное	Mимя	M*
длиной 3 ячейки	Tимя	T*
длиной 4 ячейки	Qимя	Q*
беззнаковая кодировка	Uимя	U.

Вывод, Печать

напечатать	.имя	.S
напечатать численно (имя подчеркивает тип)	имя.	D. U.
напечатать выровненным справа	имя.R	U.R

Количество

"на"	/имя	/SIDE

Последовательности

начало	<имя	<#
конец	имя>	#>

Текст

следует строка, ограниченная "	имя"	ABORT" текст"
текстовый или строковый оператор (в Бейсике -$)	"имя	"СРАВНИТЬ
массив суперстрок (superstring array)	"имя"	"ЦВЕТА"

КАК ПРОИЗНОСИТЬ СИМВОЛЫ.

! store - записать, загрузить
@ fetch - разыменовать, взять, достать
sharp, number - диез (или "число", "номер")
$ dollar - рубль
% percent - процент
^ caret - шапка, не
& ampersand - амперсанд, и
* star - звездочка
(left paren - левая скобка
) right paren - правая скобка
- dash - прочерк, минус, не
+ plus - плюс
= equals - равно
{} faces, curly brackets - фигурные скобки
[] square brackets - квадратные скобки
" quote - кавычка
' tick, prime - штрих
~ tilde - тильда
| bar - вертикальная черта
\ backslash - обратный слеш (также "под", "вниз" и "пропустить")
/ slash - слеш (также "вверх")
< less-then, left-dart - меньше чем, левая угловая скобка
> greater-then, right-dart - больше чем, правая угловая скобка
? question, query - вопрос, запрос
, comma - запятая
. dot - точка